TURBO PASCAL®
für Windows

PROGRAMMIER-
HANDBUCH

BORLAND

Turbo Pascal für Windows

Programmierhandbuch

BORLAND

Copyright © 1991 by Borland International, INC.
Jede Vervielfältigung dieses Handbuches sowie des Softwareprogrammes Turbo Pascal für Windows wird strafrechtlich verfolgt. Die Rechte an der Dokumentation und die Rechte an dem Softwareprogramm Turbo Pascal für Windows liegen bei Borland International.

Der rechtmäßige Erwerb der Programmdisketten und der Handbücher erlaubt die Nutzung der Programme analog der Benutzung eines Buches. Entsprechend der Unmöglichkeit, daß ein Buch zugleich an verschiedenen Orten von mehreren Personen gelesen wird, darf das Softwareprogramm Turbo Pascal für Windows nicht gleichzeitig von verschiedenen Personen an verschiedenen Orten und auf verschiedenen Geräten benutzt werden. Diskettenkopien dürfen lediglich zum Zweck der Datensicherung angefertigt werden.

Sie (der Lizenznehmer von Turbo Pascal für Windows) dürfen eigene Programme, die Sie mit dem Sprachcompiler Turbo Pascal für Windows geschrieben und compiliert haben, ohne Einschränkung oder Verpflichtung zu zusätzlichen Lizenzgebühren verwenden, weitergeben oder verkaufen, solange diese Programme einen Copyright-Vermerk enthalten. Unter Copyright-Vermerk verstehen wir entweder Ihren eigenen Copyright-Vermerk oder gegebenenfalls einen Hinweis wie »Created using © Borland International 1991«.

Die Beispielprogramme von Turbo Pascal für Windows, die auf den Originaldisketten mitgeliefert werden demonstrieren die verschiedenen Eigenschaften von Turbo Pascal für Windows. Sie dienen ausschließlich zu Lehrzwecken. Sie dürfen diese Beispielprogramme editieren oder für den eigenen Gebrauch modifizieren. Sie dürfen diese Programme jedoch nicht als Ganzes oder in Teilen in compilierter Form oder als Quelltext weitergeben oder verkaufen. Sie dürfen die Routinen aus den Beispielprogrammen in eigenen Anwendungsprogrammen verwenden, solange die resultierenden Programme nicht einem Bespielprogramm oder einem Teil eines Beispielprogrammes in Erscheinung oder Funktion gleichen.

Einschränkung der Gewährleistung
Es wird keine Garantie für die Richtigkeit des Inhaltes dieses Handbuches übernommen. Da sich Fehler, trotz, aller Bemühungen, nie vollständig vermeiden lassen, sind wir für Hinweise jederzeit dankbar.

Herausgeber
Borland GmbH
Lindwurmstraße 88
Postfach 15 03 40
8000 München 2
Telefon 0 89 / 72 01 00
Telefax 0 89 / 77 93 39
Telex 5 21 26 37 mcm d

Ubertragung ins Deutsche:
Marcus O. Lerch, Arne Schäpers und Borland GmbH

Typographische Konzeption des Innenteils: Rudolf Paulus Gorbach

Grafiken: Sid Gastl, Andreas Höhne

PostScript-Belichtung: CD Computer & Dokumentations GmbH, Aalen

Druck und Bindung: Schoder Druck GmbH, Gersthofen

Papier: Schleipen GmbH, Bad Dürkheim. Das verwendete Papier ist säurefrei und alterungsbeständig. Die Produktion erfolgt in einem geschlossenen Wasserkreislauf ohne Einsatz von optischen Aufhellern und unter Verwendung von chlorarm gebleichten, holzfreien Zellstoffqualitäten. Die Abwasserreinigung wird in einer betriebseigenen chemisch-biologischen Kläranlage vorgenommen.

April 1991

Inhalt

Einführung 1
Was in den Kapiteln zu finden ist 1
Typographie 3

I. Der Turbo Pascal Standard 5

1. Sprachelemente und Konstanten 7
Spezielle Symbole und reservierte Wörter 7
Bezeichner 9
Labels 10
Numerische Werte 11
Zeichenketten (Strings) 12
Deklaration von Konstanten 13
Kommentare 15
Programmzeilen 15

2. Blöcke, Lokalität und Gültigkeitsbereich 17
Syntax 17
Gültigkeitsbereiche 19
Neudeklaration in einem untergeordneten Block 19
Der Ort einer Deklaration in einem Block 20
Neudeklaration innerhalb eines Blocks 20
Standardbezeichner 20
Der Gültigkeitsbereich von Bezeichnern im Interface-Teil 20

3. Typen 23
Einfache Typen 24
Ordinale Typen 24
Realzahlen 29
Der Typ String 30
Strukturierte Typen 30
Array-Typen 31

Record-Typen 32
Objekttypen 34
Mengen-Typen 40
Dateitypen 41
Zeiger-Typen 42
Prozedurtypen 43
Die Deklaration eines Prozedurtyps 43
Identische und kompatible Typen 44
Identische Typen 44
Kompatibilität von Typen 45
Zuweisungs-Kompatibilität 45
Der Typen-Definitionsteil 46

4. Variablen 49

Die Deklaration von Variablen 49
Das Daten-Segment 50
Das Stack-Segment 50
Absolute Variablen 51
Der Bezug auf Variablen 52
Die Festlegung eines Variablenbezugs 52
Arrays, Strings und Indizierungen 53
Records und Feldbezeichner 54
Bezeichner von Objektkomponenten 55
Zeiger und dynamische Variablen 55
Umwandlung von Variablentypen 55

5. Typisierte Konstanten 59

Einfache typisierte Konstanten 60
Strings 60
Strukturierte Typen 61
Arrays 61
Records 62
Konstanten mit Objekttypen 63
Mengen 64
Zeiger 64
Prozedurale Typen 65

6. Ausdrücke 67

Syntax von Ausdrücken 67
Operatoren 71
Arithmetische Operatoren 71
Logische Operatoren 72
Boolesche Operatoren 73
String-Operatoren 74

PChar-Operatoren 74
Mengen-Operatoren 75
Relationale Operatoren 76
Der Adreß-Operator @ 77
Funktionsaufrufe 80
Mengen-Konstruktoren 81
Umwandlung von Wert-Typen 81
Prozedur-Variablen in Ausdrücken 82

7. Anweisungen 85

Einfache Anweisungen 85
Zuweisungen 86
Prozeduraufrufe 87
Die Anweisung goto 88
Strukturierte Anweisungen 89
Verbundanweisungen 89
Bedingte Anweisungen 90
Schleifen 92
Die with-Anweisung 96

8. Prozeduren und Funktionen 99

Prozedur-Deklarationen 99
Near- und far-Deklarationen 100
Deklarationenmit export 101
Deklarationenmit interrupt 101
Deklarationen mit forward 102
Deklarationen mit external 103
Deklarationenmit assembler 104
Deklarationen mit inline 104
Funktions-Deklarationen 105
Deklarationen von Methoden 106
Konstruktoren und Destruktoren 108
Parameter 109
Wertparameter 110
Variablenparameter 110
Untypisierte Variablenparameter 111
Prozedurtypen 112
Prozedur-Variablen 112
Prozedur-Variablen in Strukturen 115
Prozedur-Variablen als Parameter 115

9. Programme und Units 119

Die Syntax eines Programms 119
Der Programmkopf 119

Die Anweisung uses 120
Die Syntax von Units 120
Der Kopf einer Unit 121
Der Interface-Teil 121
Der Implementations-Teil 121
Der Initialisierungs-Teil 123
Units, die andere Units voraussetzen 123
uses-Anweisungen im Implementationsteil und gegenseitige Abhängigkeiten 124

10. Dynamische Linkbibliotheken 127

Was ist eine DLL? 127
Verwendung von DLLs 128
In Units importieren 129
Statischer und dynamischer Import 130
Erstellen von DLLs 131
Die Deklaration export 133
Die Anweisung exports 133
Initialisierungs- und Austrittscode der Bibliothek 134
Hinweise zur Programmierung einer Bibliothek 136
Globale Variablen in einer DLL 136
Globaler Speicher und Dateien in einer DLL 136
DLLs und die Unit System 137
Laufzeitfehler in DLLs 137
DLLs und Stacksegmente 138

II. Die Standard-Units 139

11. Die Unit System 141

Standardprozeduren und -funktionen 141
Ein- und Ausgabe 144
Grundlegendes 145
Standard-Prozeduren und Funktionen für Textdateien 146
Standard-Prozedurenfür untypisierte Dateien 148
Die Variable FileMode 149
Geräte in Turbo Pascal 149
Vordefinierte Variablen 150

12. Die Unit WinDos 153

Konstanten, Typen und Variablen 153
Typen 155
Variablen 157
Prozeduren und Funktionen 157

13. Die Unit Strings 161

Was ist ein null-terminierter String? 161
Verwendung von null-terminierten Strings 161
Zeichenzeiger und String-Literale 162
Zeiger auf Zeichen und Zeichenarrays 163
Indizieren von Zeichenzeigern 164
Operationenmit Zeigern 165
Null-terminierte Strings und Standard-Prozeduren 166
Was die Unit Strings definiert 166

14. Die Unit WinCrt 171

Einbindung der Unit WinCrt 171
Sonderzeichen 172
Zeilen-Input 173
Variablen 173
Prozeduren und Funktionen 176

15. Der Coprozessor 8087 181

Die Datentypen des 8087 182
Arithmetische Operationen im Format Extended 183
Vergleiche von Realzahlen 184
Der Rechenstack des 8087 185
Die Ausgabe von Realzahlen 186
Units und der 8087 186
Ermittlungdes 8087 187
Maschinenprogramme und der Emulator 187

III. Turbo Pascal Internas 189

16. Interne Details 191

Code-Segmente 191
Segmentattribute 191
Verändern von Attributen 192
Das automatische Datensegment 192
Der Heap-Manager 194
Die Variable HeapError 195
Interne Datenformate 196
Integer-Typen 196
Char-Typen 196
Die Typen Boolean, WordBool und LongBool 197
Aufzählungstypen 197
Real-Typen 197
Zeiger-Typen 200

String-Typen 200
Mengen 201
Array-Typen 201
Record-Typen 201
Datei-Typen 201
Prozedurale Typen 203
Direkte Zugriffe auf Speicher-Adressen 203

17. Objekte 205

Das interne Datenformat von Objekten 205
Virtuelle Methodentabellen 206
DynamischeMethodentabellen 208
Die Standardfunktion SizeOf 211
Die Standardfunktion TypeOf 211
Der Aufruf von virtuellen Methoden 211
Der Aufruf von dynamischen Methoden 212
Aufrufkonventionen von Methoden 213
Konstruktoren und Destruktoren 213
Erweiterungen von New und Dispose 214
Methoden in Assembler 216
Fehlerbehandlung in Konstruktoren 219

18. Interne Details 223

Der Aufruf von Funktionen und Prozeduren 223
Variablen-Parameter 223
Wert-Parameter 224
Funktionsergebnisse 225
Aufrufe als near und far 225
Eingeschachtelte Routinen 226
Start- und Endcode von Prozeduren und Funktionen 227
Verfügbare Prozessor-Register 229
Exit-Prozeduren 229
Bearbeitung von Interrupts 231
Interrupt-Prozeduren 231

19. Ein- und Ausgabe 233

Textdatei-Gerätetreiber 233
Die Open-Funktion 234
Die InOut-Funktion 234
Die Flush-Funktion 235
Die Close-Funktion 235
Direkte Zugriffe auf I/O-Adressen 236
Die Arrays Port und PortW 236

20. Automatische Optimierung 237

Vorausberechnungvon Konstanten 237
Gemeinsame Speicherungmehrfach verwendeter Konstanten 237
Auswertung zusammengesetzter boolescher Ausdrücke 238
Reihenfolge der Berechnung 239
Überprüfung vonWertebereichen 239
Schiebebefehle anstelle von Multiplikationen 240
Wortweise Ausrichtungvon Daten 240
Entfernung »toten« Codes 240
Intelligentes Linken 241

21. Compiler-Befehle 243

Schalter 244
Parameter 250
Bedingte Compilierung 253
Symbole zur bedingten Compilierung 254

IV. Turbo Pascal und Assembler 257

22. Der integrierte Assembler 259

Die Anweisung asm 259
Einsatz der Register 260
Die Syntax einer Assembler-Anweisung 261
Labels 261
Präfix-Opcodes 262
Befehlssätze 263
Assembler-Anweisungen 265
Operanden 266
Ausdrücke 267
Unterschiede zwischen Pascal- und Assembler-Ausdrücken 267
Ausdruck-Elemente 269
Ausdruck-Klassen 274
Ausdruck-Typen 275
Ausdruck-Operatoren 277
Assembler-Prozeduren und Funktionen 279

23. Einbinden von Assembler-Code 283

Maschinenprogramme mit dem Turbo Assembler 284
Einige Beispiele 285
Ein weiterer Ausblick 288
Maschinencode mit inline 289
inline-Anweisungen 289
inline-Deklarationen 291

V. Die Turbo Pascal Referenz 293

24. Verzeichnis der Routinen 295

VI. Anhang 375

A. Fehlermeldungen und -codes 377

Fehlermeldungen des Compilers 377
Laufzeitfehler 395
Fehlercodes von DOS 395
Ein-/Ausgabefehler 397
Fehler mit sofortigem Abbruch (»Fatal Errors«) 398

B. Zeichensatz und Tastaturcodes 401

ASCII-Codes 401
Scancodes der Tastatur 406

Index 409

Einführung

Dieses Handbuch ist für erfahrene Programmierer zum Nachschlagen gedacht. Wenn Sie bereits einer Programmiersprache mächtig sind, am besten natürlich Pascal, ist dieses Buch für Sie genau das richtige. Es enthält die Referenz zum Turbo-Pascal-Standard, zusammenfassende Informationen zur Laufzeitbibliothek sowie Details zu den internen Abläufen eines Turbo-Pascal-Programms und Informationen darüber, wie in Assembler geschriebene Routinen in Turbo-Pascal-Programme eingebunden werden können. Wenn Sie

- noch nie programmiert haben, gleich mit welcher Programmiersprache,
- programmieren können, aber gerne Pascal lernen möchten,
- in Pascal programmieren können, aber die integrierte Entwicklungsumgebung kennenlernen möchten,

dann lesen Sie bitte nicht diese Dokumentation, sondern das Benutzerhandbuch.

Was in den Kapiteln zu finden ist

Das Programmierhandbuch ist in sechs Teile gegliedert. Der erste Teil beschreibt den Turbo-Pascal-Standard und erstreckt sich über die folgenden Kapitel:

Kapitel 1 bis 9 »Sprachelemente«, »Blöcke«, »Typen«, »Variablen«, »Typisierte Konstanten«, »Ausdrücke«, »Anweisungen«, »Prozeduren und Funktionen«, »Programme und Units«.

Kapitel 10 »Dynamische Linkbibliotheken« beschreibt, wie Sie mit Turbo Pascal für Windows DLLs schreiben und für Ihre Programme nützen können.

Der zweite Teil enthält in den Kapiteln 11 bis 14 Informationen zu den Standard-Units *System*, *WinDos*, *Strings* und *WinCrt*. Abgeschlossen wird dieser Teil mit dem Kapitel 15 »Der Coprozessor 8087«, der auf die Nutzung des numerischen Coprozessors unter Turbo Pascal eingeht. Hier finden Sie detaillierte Informationen über die Datentypen *Single*, *Double*, *Extended* und *Comp*,

die Turbo Pascal im Modus {$N+} zur Verfügung stellt – auch wenn der Computer nicht mit einem 8087, 80287 oder 80387 ausgerüstet ist.

Der dritte Teil bietet technische Informationen speziell für fortgeschrittene Benutzer:

Kapitel 16 »Speicherbelegung und Datenformate« stellt dar, wie Turbo Pascal den Speicher verwaltet, und wie die vordefinierten Datentypen gespeichert werden.

Kapitel 17 »Objekte« beschreibt, wie Turbo Pascal vorgeht, um virtuelle Methoden dynamisch zu binden, und zeigt, wie Fehler bei der Initialisierung von Objekten abgefangen werden können.

Kapitel 18 »Interne Details« geht auf die Aufrufkonventionen von Routinen ein, die Turbo Pascal befolgt, und auf die Definition von eigener Exit-Prozeduren und Interrupt-Routinen.

Kapitel 19 »Ein- und Ausgabe« zeigt, wie Sie in Turbo Pascal eigene Gerätetreiber schreiben können.

Kapitel 20 »Automatische Optimierung« beschreibt die Verfahren, mittels derer der Compiler den erzeugten Code optimiert.

Kapitel 21 »Compiler-Befehle« listet und erläutert die Compiler-Befehle, die den Ablauf der Compilierung steuern.

Der vierte Teil ist für jene Benutzer besonders interessant, die Ihre Programme hinsichtlich Laufzeit und Codegröße optimieren wollen:

Kapitel 22 »Der integrierte Assembler« beschreibt Funktion und Möglichkeiten des in Turbo Pascal für Windows integrierten Assemblers.

Kapitel 23 »Einbinden von Assembler-Code« erläutert die Voraussetzungen, die in Assembler codierte Routinen erfüllen müssen, damit sie in Pascal-Programme eingebunden werden können.

Der fünfte Teil umfaßt nur das Kapitel 24 »Verzeichnis der Routinen«. Hier werden sämtliche Prozeduren und Funktionen der Units *System*, *Strings*, *WinDos* und *WinCrt* in alphabetischer Reihenfolge detailliert besprochen.

Der sechste und letzte Teil schließlich besteht aus zwei Anhängen:

Anhang A »Fehlermeldungen und -codes« erläutert sowohl die Fehlermeldungen, die bei der Compilierung eines Programms eventuell ausgegeben werden, als auch die Codes, die beim Abbruch eines Programms auf den aufgetretenen Fehler hinweisen.

Anhang B »Zeichensatz und Tastaturcodes« besteht aus einer tabellarischen Auflistung sowohl des IBM- als auch des ANSI-Zeichensatzes, der unter Windows gültig ist, und der von der Tastatur zurückgegebenen Codes.

Typographie

Zur Verdeutlichung von Zusammenhängen verwenden wir mehrere unterschiedliche Schriftarten:

`Schreibmaschine` steht für Syntaxdefinitionen, Eingaben des Benutzers, Ausgaben von Beispielprogrammen und wird für Programmlistings verwendet.

`Schreibmaschine fett` kennzeichnet in Listings reservierte Wörter von Turbo Pascal, zum Beispiel `procedure`, `begin`, `object` usw.

Fettschrift wird im laufenden Text benutzt, um reservierte Wörter (wie **begin**, **end**, **object** usw.) hervorzuheben

Kursivschrift kennzeichnen Namen von Variablen und Bezeichnern im laufenden Text. Außerdem dient diese Schriftart zur Hervorhebung neuer Begriffe.

`Tasten` für die Tasten Ihrer Tastatur, zum Beispiel in »Ein Druck auf `Esc` entfernt das Menü wieder vom Bildschirm«.

I. Der Turbo Pascal Standard

1. Sprachelemente und Konstanten

Sprachelemente (Tokens) sind die kleinsten bedeutungstragenden Einheiten eines Programms. Sie werden in die Kategorien spezielle Symbole, Bezeichner, Labels, Zahlen und Zeichenketten unterteilt.

Ein Pascal-Programm besteht aus Sprachelementen und Begrenzern, die die einzelnen Elemente voneinander trennen. Ein Begrenzer kann entweder ein Leerzeichen, ein Kommentar oder ein Zeilenvorschub sein. Zwei aufeinanderfolgende Sprachelemente müssen durch mindestens einen Begrenzer voneinander getrennt sein, wenn es sich bei beiden Sprachelementen um reservierte Wörter, Bezeichner, Labels oder Zahlen handelt.

Ein Begrenzer kann nicht Teil eines Sprachelements sein. Ausnahme: Das Sprachelement »Zeichenkette« kann Zeichen enthalten, die ansonsten als Begrenzer verwendet werden.

Spezielle Symbole und reservierte Wörter

Turbo Pascal benutzt die folgenden Teilbereiche des ASCII-Zeichensatzes:

- *Buchstaben* – A bis Z und a bis z. Die deutschen Umlaute und das »ß« sind nicht zulässig;
- *Ziffern* – die arabischen Ziffern von 0 bis 9;
- *Hexadezimalziffern* – die arabischen Zahlen von 1 bis 9 sowie die Buchstaben A bis F und a bis f;
- *Leerzeichen* – das Leerzeichen selber (ASCII-Wert 32) und alle Steuerzeichen (ASCII 0 - 31) einschließlich des Zeilenendezeichens (ASCII 13).

Spezielle Symbole und reservierte Wörter haben eine festgelegte Bedeutung (in einigen Fällen auch mehrere, abhängig vom Kontext). Die folgenden Zeichen werden von Turbo Pascal als spezielle Symbole interpretiert:

```
+ - * / = < > [ ] . , ( ) : ; ^ @ { } $ #
```

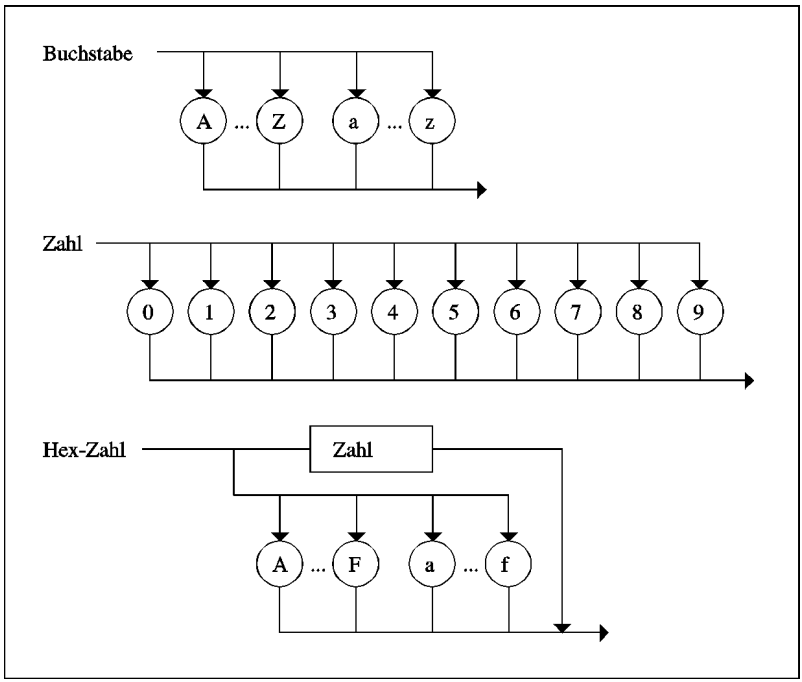


Abbildung 1.1 Von Turbo Pascal verwendete ASCII-Zeichen

Zusätzlich werden auch noch einige Paare von Spezialsymbolen als Operatoren und/oder Begrenzer verwendet:

:=	Zuweisungsoperator
<> <= >=	relationale Operatoren
..	Teilbereiche
(. .)	kann als Ersatz für [ ] verwendet werden
(* *)	kann anstelle von { } verwendet werden

Eine Unterscheidung zwischen Groß- und Kleinschreibung findet nicht statt – reservierte Wörter und alle anderen Bezeichner können also beliebig geschrieben werden.

Turbo Pascal reserviert die folgenden Wörter, die in diesem Handbuch alle durch **Fettdruck** hervorgehoben sind:

and	**array**	**asm**	**begin**	**case**
const	**constructor**	**destructor**	**div**	**do**
downto	**else**	**end**	**exports**	**file**
for	**function**	**goto**	**if**	**implementation**
in	**inline**	**interface**	**label**	**library**
mod	**nil**	**not**	**object**	**of**
or	**packed**	**procedure**	**program**	**record**
repeat	**set**	**shl**	**shr**	**string**

then	**to**	**type**	**unit**	**until**
uses	**var**	**while**	**with**	**xor**

Zusätzlich definiert Turbo Pascal die Standard-Direktiven

absolute	**assembler**	**export**	**external**
far	**forward**	**index**	**interrupt**
name	**near**	**private**	**resident**
virtual			

Anders als reservierte Wörter können Standard-Direktiven vom Benutzer neu definiert werden. Dies sollte man jedoch vermeiden. *private* ist innerhalb einer Objekttyp-Deklaration ein reserviertes Wort.

Bezeichner

Alle Typen, Variablen, Prozeduren, Funktionen, Units, Programme und Felder von Records werden durch Bezeichner identifiziert. Ein Bezeichner kann beliebig lang sein, allerdings sind nur die ersten 63 Zeichen signifikant.

Ein Hinweis zu den Syntaxdiagrammen: Die Namen innerhalb der Kästen und Kreise stehen für die Elemente, aus denen sich ein Konstrukt zusammensetzt und sind durch Pfeile miteinander verbunden. Um ein gültiges Konstrukt zu

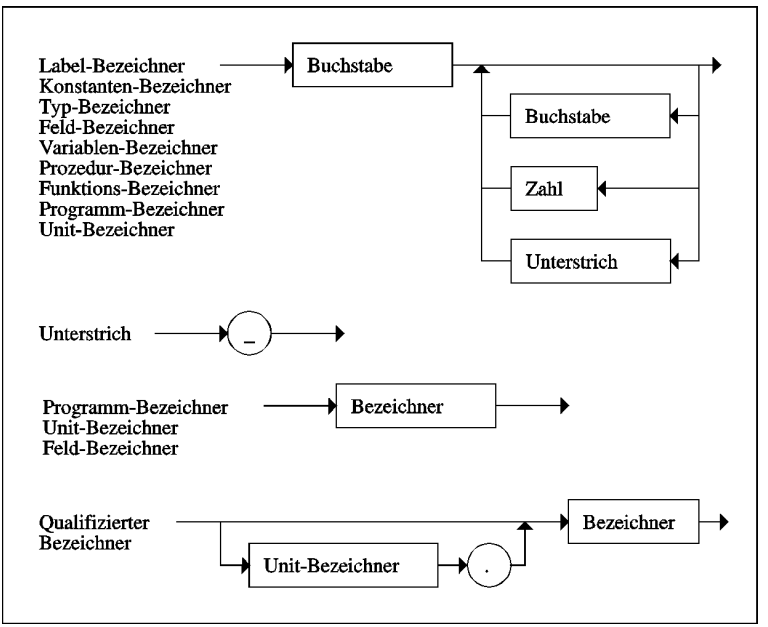


Abbildung 1.2 Syntax von Standardbezeichnern

erhalten, beginnen Sie links im Diagramm, folgen den Pfeilen (die teilweise mehrere Möglichkeiten und/oder Wiederholungen zulassen) und »addieren« die einzelnen Elemente – solange, bis Sie die rechte Seite des Diagramms erreicht haben.

Ein Bezeichner muß mit einem Buchstaben oder einem Unterstrich (ASCII-Code $5F) beginnen und darf keine Leerzeichen enthalten. Auf das erste Zeichen können Ziffern, Buchstaben und Unterstriche folgen. Bei Bezeichnern macht Turbo Pascal ebenfalls keinen Unterschied zwischen Groß- und Kleinschreibung, die deutschen Umlaute und das »ß« sind nicht zulässig.

Der Bezug auf Bezeichner gleichen Namens, die innerhalb verschiedener Units (siehe Kapitel 9 »Programme und Units«) deklariert sind, geschieht über eine *Qualifizierung* durch Voranstellen des Unitnamens. Einige Beispiele für zulässige Bezeichner:

```
Writeln
Exit
Real2String
System.MemAvail
Strings.StrLen
WinCrt.ReadText
```

In diesem Handbuch sind alle Bezeichner *kursiv* gesetzt, sofern sie im Text erwähnt werden.

Labels

Ein Label ist eine Zahlenfolge im Wertebereich von 0 bis 9999; führende Nullen sind nicht signifikant. Labels werden ausschließlich in Zusammenhang mit der Anweisung **goto** verwendet.

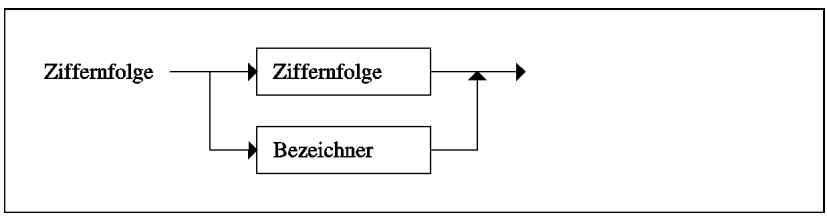


Abbildung 1.3 Syntax eines Labels

Als Erweiterung von Standard-Pascal erlaubt Turbo Pascal die Benutzung von Bezeichnern als Labels.

Numerische Werte

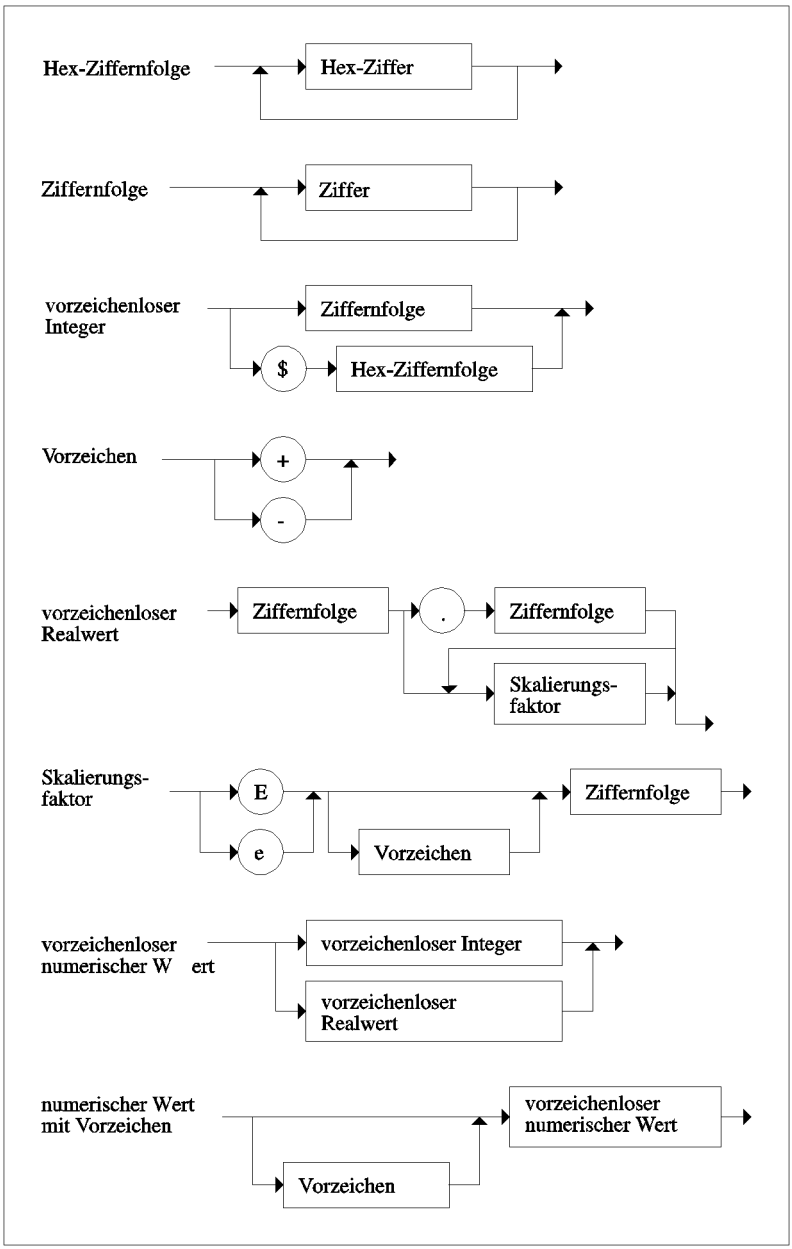


Abbildung 1.4 Syntax numerischer Werte

Wertangaben können durch die folgenden Syntaxdiagramme formal dargestellt werden:

Für die Angabe konstanter Integer- und Realwerte wird die normale dezimale Schreibweise verwendet. Hexadezimalen Integer-Konstanten wird zur Kennzeichnung ein Dollarzeichen ($) vorangestellt. Die wissenschaftliche Notation (E oder e, gefolgt von einem Exponenten) wird als »mal 10 hoch n« gelesen. Einige Beispiele: 7E-2 steht für $7 * 10^{-2}$; 12.25e6 und 12.25e+6 sind äquivalent und stehen für $12.25 * 10^{+6}$.

Zahlen mit Nachkommastellen oder Exponenten werden grundsätzlich als Realwerte behandelt, ganzzahlige Dezimalwerte werden als Integer gespeichert und müssen im Bereich von -2147483648 bis 2147483647 liegen. Hexadezimalzahlen sind ihrer Natur nach vorzeichenlos, d.h. das Vorzeichen ergibt sich aus der Größe des angegebenen Wertes, der zwischen $00000000 und $FFFFFFFF liegen muß.

Zeichenketten (Strings)

Eine Stringkonstante ist eine Folge von Zeichen des erweiterten Zeichensatzes (siehe Anhang B »Zeichensatz und Tastaturcodes«), die durch Apostrophe begrenzt ist und innerhalb des Quelltextes in einer Zeile stehen muß. Ein String kann auch leer sein, d.h. zwischen den beiden Begrenzern müssen nicht unbedingt weitere Zeichen stehen. Ein Apostrophe *innerhalb* eines Strings läßt sich durch zwei aufeinanderfolgende Apostrophe darstellen.

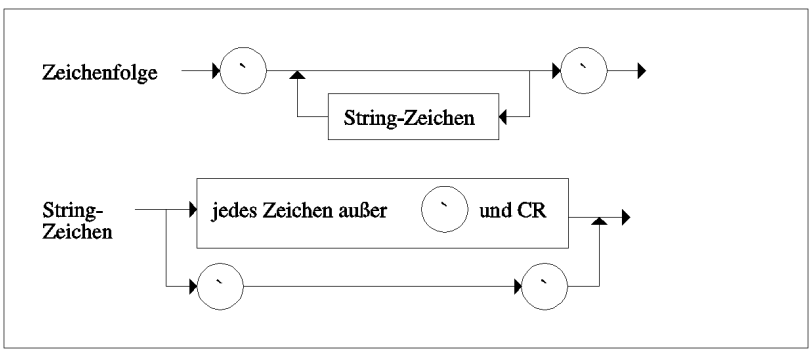


Abbildung 1.5 Syntax eines Strings

Als Erweiterung des Standards erlaubt Turbo Pascal die Aufnahme von Steuer- und Grafikzeichen in Strings: das Symbol #, gefolgt von einer vorzeichenlosen Integerkonstanten im Bereich von 0 bis 255, wird als ein Zeichen des entsprechenden ASCII-Codes interpretiert. Zwischen dem # und der Integerkonstanten dürfen keine weiteren Zeichen stehen; mehrere aufeinanderfolgende Steuerzei-

chen (wie z.B. #13#10 für CR/LF) werden nicht durch zusätzliche Begrenzer voneinander getrennt.

Die Länge eines Strings ergibt sich aus der Anzahl der tatsächlich enthaltenen Zeichen: ein String mit dem Inhalt #65 hat also dieselbe Länge wie 'A', nämlich 1. Ein String von der Länge Null (eine leere Zeichenkette) ist nur zu String-Typen, ein String mit der Länge von einem Zeichen dagegen zu String- und Char-Typen kompatibel. Ein String mit mehr als einem Zeichen ist zu jedem String-Typ und jedem gepackten String-Typ kompatibel, darüber hinaus auch zu dem Datentyp *PChar*, wenn die erweiterte Syntax eingeschaltet **{$X+}** ist. Die folgenden Beispiele sind gültige Angaben von Strings:

```
'Turbo'   'Wie geht''s'   ''''   ';'   ' '
#13#10   'Zeile1'#13'Zeile2'   #7#7'Aufwachen!'#7#7
```

Deklaration von Konstanten

Eine Konstanten-Deklaration vereinbart einen Bezeichner, der innerhalb des entsprechenden Blocks für einen konstanten Wert steht. Die Wertangabe für eine Konstante kann nicht rückbezüglich sein, d.h. die Konstante selbst darf innerhalb der Wertangabe nicht erscheinen.

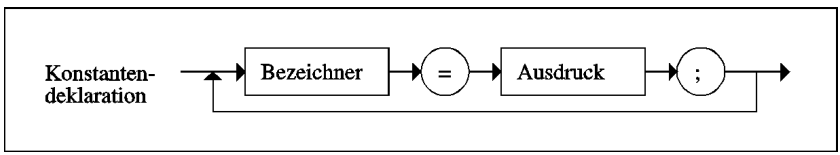


Abbildung 1.6 Syntax einer Konstanten-Deklaration

Vorzeichenbehaftete Konstanten können nur für einen Integer- oder Realwert stehen.

Während der Compiler im Standard-Pascal keine Berechnung von Konstanten vornimmt und sich **const**-Deklarationen folglich auf direkte Angaben wie

```
const
  BUFMAX = 100;
var
  Buf: array[0..99] of Integer;  { [0..BUFMAX-1] ist nicht erlaubt! }
```

beschränken müssen, erlaubt Turbo Pascal Konstanten-*Ausdrücke* an jeder Stelle, an denen der Sprachdefinition nach nur eine einfache Konstante möglich ist. Damit lassen sich nicht nur Deklarationen wesentlich übersichtlicher schreiben:

```
const
  BUFMAX = 100;
```

```
var
  Buf: array[0..BUFMAX-1] of Integer;  { entspricht [0..99] }
```

Einige weitere Beispiele dazu:

```
const
  Min      = 0; Max      = 100;
  Center   = (Max - Min) div 2;

  Beta     = chr(255);
  NumChars = Ord('Z') - Ord('A') + 1;

  Message  = 'Kein Speicherplatz!';
  ErrMsg   = 'Fehler: '+ Message;
  ErrPos   = 80 - Length(ErrMsg) div 2;
  Ln10     = 2.3025890;
  Ln10R    = 1 / Ln10;

  Numset   = ['0'..'9'];   Charset  = ['A'..'Z'];
  Alphaset = Numset + Charset;
```

Da der Compiler den Wert eines Konstanten-Ausdrucks während der Übersetzung bestimmen muß, gibt es natürlich auch einige Einschränkungen, die sich größtenteils selbst erklären sollten. Die folgenden Elemente dürfen in einem derartigen Ausdruck *nicht* enthalten sein:

- Werte von Variablen und typisierten Konstanten (außer von konstanten Adreß-Ausdrücken, vgl. Kapitel 5 »Typisierte Konstanten«);
- der Adreß-Operator @ (außer bei konstanten Adreß-Ausdrücken, vgl. Kapitel 5 »Typisierte Konstanten«);
- Aufrufe von Funktionen. Wie zuvor teilweise demonstriert, gibt es hier allerdings eine ganze Reihe von Ausnahmen:

```
Abs    Chr   Hi      Length   Lo     Odd    Ord
Pred   Ptr   Round   SizeOf   Succ   Swap   Trunc
```

Abgesehen von diesen Einschränkungen folgen Konstanten-Ausdrücke denselben Regeln wie in Kapitel 6 »Ausdrücke« beschrieben. Sie sind folglich auch nicht auf reine **const**-Deklarationen beschränkt:

```
const
  BufSize = 512;

type
  CharVal = Ord('A')..('Z');   { eine Untermenge }
  RecPtr  = ^DataRec;
  DataRec = record
              Prev, Next: RecPtr;
              Data: string[15];
            end;
  RecBuf  = array[0..SizeOf(DataRec)-1] of Byte;
```

Kommentare

Die beiden folgenden Konstrukte werden vom Compiler als Kommentar behandelt und bei der Compilierung ignoriert:

```
{dieser Satz steht in geschweiften Klammern}
(* das ist eine Klammer mit einem Stern *)
```

Kommentare, die als erstes Zeichen nach { bzw. (* ein Dollarzeichen ($) enthalten, werden als Compiler-Befehle interpretiert (vgl. Kapitel 21 »Compiler-Befehle«). Beispiele:

```
{$N+}   { setzt den Modus N+ (IEEE-Typen) }
(*$N+*) { hat denselben Effekt }
```

Programmzeilen

In Turbo Pascal darf die Länge einer Programmzeile 126 Zeichen nicht überschreiten.

2. Blöcke, Lokalität und Gültigkeitsbereich

Ein Block besteht aus beliebig angeordneten Deklarationen und Anweisungen. Jeder Block ist Teil einer Prozedur-Deklaration, einer Funktions-Deklaration, eines Programms oder einer Units. Alle Bezeichner und Labels im Deklarationsteil des Blocks sind in ihrer Wirkung auf diesen Block beschränkt – sie sind *lokal* zu diesem Block.

Syntax

Der syntaktische Aufbau jedes Blocks läßt sich durch folgendes Diagramm darstellen:

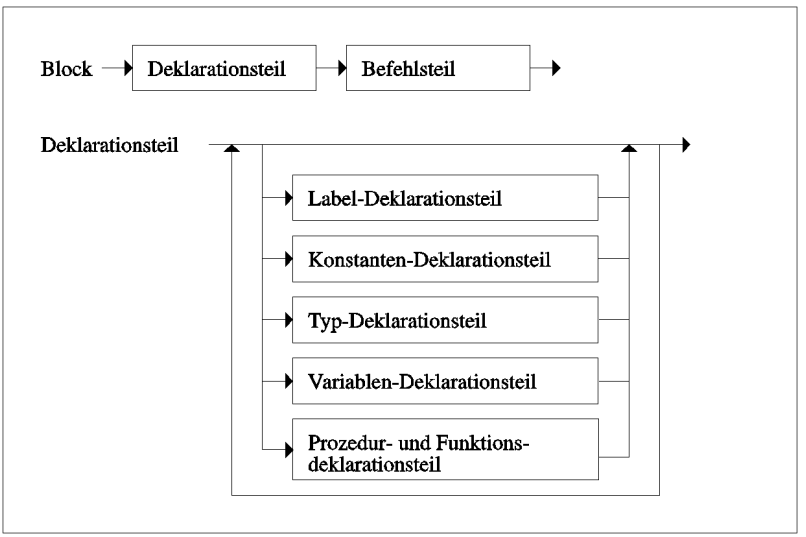


Abbildung 2.1 Syntaktischer Aufbau von Blöcken

Im Label-Deklarationsteil werden alle Labels vereinbart, die **goto**-Sprungziele im Anweisungsteil des betreffenden Blocks darstellen sollen. Jedes Label darf innerhalb des Anweisungsteils nur einmal definiert werden (d.h. jedes `goto` muß ein eindeutiges Ziel haben). Ziffernfolgen, die für Labels verwendet werden, müssen im Bereich von 0 bis 9999 liegen.

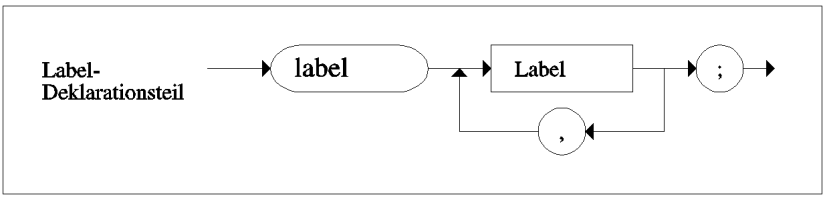


Abbildung 2.2 Syntax des Deklarationsteils eines Labels

Der *Deklarationsteil für Konstanten* beinhaltet alle Vereinbarungen von Konstanten, die lokal zum entsprechenden Block sind.

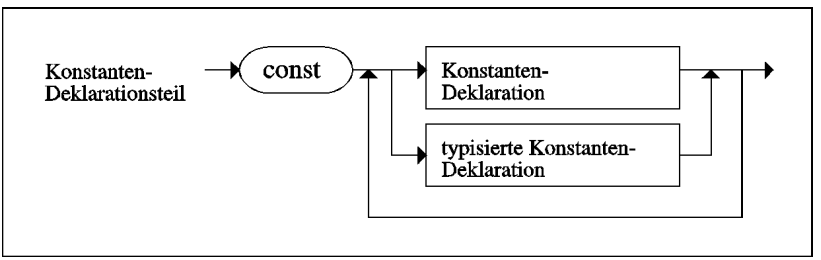


Abbildung 2.3 Syntax des Deklarationsteils für Konstanten

Der *Deklarationsteil für Typen* beinhaltet alle lokalen Typendefinitionen des dazugehörigen Blocks.

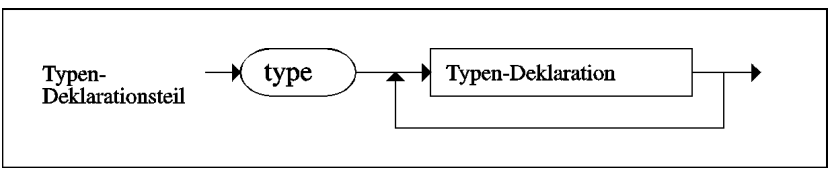


Abbildung 2.4 Syntax des Deklarationsteils für Typen

Der *Deklarationsteil für Variablen* beinhaltet alle Variablen-Deklarationen, die lokal zum entsprechenden Block sind.

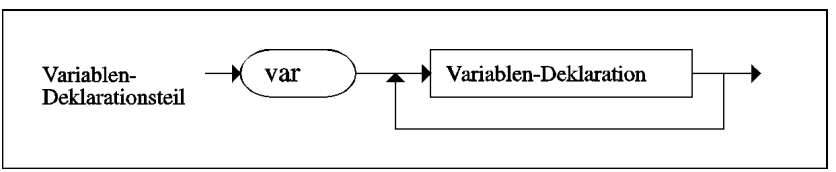


Abbildung 2.5 Syntax des Deklarationsteils für Variablen

Der *Deklarationsteil für Prozeduren und Funktionen* beinhaltet alle Prozedur- und Funktionsdeklarationen, die lokal zum entsprechenden Block sind.

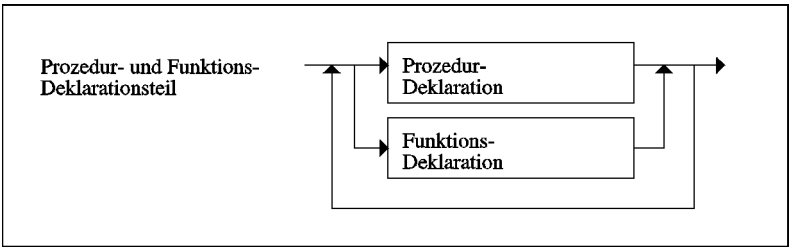


Abbildung 2.6 Syntax des Deklarationsteils für Prozeduren und Funktionen

Im *Anweisungsteil* werden schließlich alle Operationen definiert, die bei der Aktivierung des Blocks ausgeführt werden sollen.

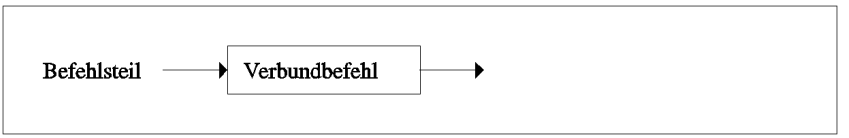


Abbildung 2.7 Syntax des Anweisungsteils

Gültigkeitsbereiche

Jeder Bezeichner und jedes Label in einer Deklaration vereinbart genau ein Objekt bzw. ein Sprungziel. Deshalb muß ein Bezeichner, genauso wie ein Label, immer im Gültigkeitsbereich (*Scope*) seiner Deklaration sein, wenn er im Programm erscheint. Der Gültigkeitsbereich von Bezeichnern und Labels liegt zwischen der eigentlichen Deklaration und dem Ende des dazugehörigen Blocks, wobei alle Blöcke mit eingeschlossen sind, die dieser Block umfaßt. Allerdings gibt es hierzu einige Ausnahmen, die in den folgenden Absätzen besprochen werden.

Neudeklaration in einem untergeordneten Block

Wenn wir davon ausgehen, daß der Block »Außen« den Block »Innen« umfaßt, d.h. ihm übergeordnet ist, dann beschränkt jede Neudeklaration eines Bezeichners von »Außen« im Block »Innen« den Gültigkeitsbereich dieses Bezeichners auf den Block »Innen« und eventuell weitere von »Innen« eingeschlossenen Blöcke. Anders gesagt: Wenn »Außen« eine Variable *x* deklariert

und »Innen« eine Variable gleichen Namens, dann können Anweisungen im Block »Innen« nicht auf die in »Außen« deklarierte Variable *x* zugreifen.

Der Ort einer Deklaration in einem Block

Bezeichner und Labels müssen deklariert sein, bevor sie in einem Block benutzt werden können. Auf Zugriffsversuche vor ihrer eigentlichen Deklaration reagiert der Compiler mit der Meldung `Unbekannter Bezeichner`. Zu dieser Regel gibt es allerdings eine Ausnahme: Der Grundtyp eines Zeigers kann tatsächlich ein noch nicht deklarierter Bezeichner sein. Aber auch dieser Bezeichner muß schließlich in derselben Typ-Deklaration erscheinen, in der auch der Zeiger steht.

Neudeklaration innerhalb eines Blocks

Bezeichner und Labels können auf der obersten Ebene eines Blocks nur jeweils einmal deklariert werden, es sei denn, ihre Neudeklaration erfolgt innerhalb eines untergeordneten Blocks oder innerhalb der Feldliste eines Records.

Der Bezeichner eines Feldes in einem Record wird innerhalb eines Record-Typs deklariert. Er ist nur dann signifikant, wenn er in Verbindung mit einem Verweis auf eine Variable desselben Record-Typs aufgeführt wird. Man kann also den Bezeichner eines Feldes innerhalb des gleichen Blocks (mit exakt demselben Namen) neu deklarieren, solange er nicht auf derselben Ebene innerhalb desselben Record-Typs steht. Ein bereits deklarierter Bezeichner kann im selben Block als Feldbezeichner erneut deklariert werden.

Der Gültigkeitsbereich einer Objekttyp-Komponente (Feld, Methode, Konstruktor, Destruktor) geht über deren Deklarationsort hinaus. Detaillierte Informationen dazu finden Sie in Kapitel 3 »Typen«.

Standardbezeichner

Turbo Pascal bietet eine ganze Reihe vordefinierter Konstanten, Typen, Prozeduren und Funktionen, die so arbeiten, als wären sie innerhalb eines Blocks deklariert worden, der das ganze Programm umschließt. Folglich umfaßt ihr Gültigkeitsbereich das gesamte Programm.

Der Gültigkeitsbereich von Bezeichnern im Interface-Teil

Sämtliche »öffentlichen« Bezeichner einer Unit, die durch die Anweisung **uses** in ein Programm oder den Interface-Teil einer anderen Unit aufgenommen

wird, stehen dem gesamten Programm bzw. der Unit zur Verfügung, sind also global.

Per Definition stellt jede via **uses** aufgenommene Unit einen Block dar, der das folgende Programm und folgende angegebene Units umschließt. Die als erste aufgeführte Unit entspricht folglich dem »äußersten« Block, die zuletzt aufgeführte Unit dem innersten Block. Daraus folgt: wenn zwei oder mehrere Units denselben Bezeichner deklarieren, beziehen sich »unqualifizierte« Referenzen (d.h. Angaben des Bezeichners ohne vorangestellten Unitnamen) auf den Bezeichner der zuletzt aufgeführten Unit – der Mechanismus ist also derselbe wie bei einer lokalen Redeklaration innerhalb des Programms.

Da die Unit *System*, in der alle vordefinierten Bezeichner von Turbo Pascal enthalten sind, implizit als allererste aufgenommen wird, können diese Bezeichner an jeder anderen Stelle redeklariert werden. Der Zugriff auf *System* ist über ein Voranstellen des Unitnamens dennoch jederzeit möglich: eine Angabe wie *System.WriteLn* ruft die vordefinierte Routine *WriteLn* auf – unabhängig davon, ob dieser Bezeichner an anderer Stelle redefiniert worden ist.

3. Typen

Jede Deklaration einer Variablen muß den Typ dieser Variablen angeben. Der Typ legt den Wertebereich der Variablen fest und bestimmt die Operationen, die mit ihr ausgeführt werden können. Eine Typen-Definition vereinbart also einen Bezeichner, der seinerseits für einen bestimmten Typ steht.

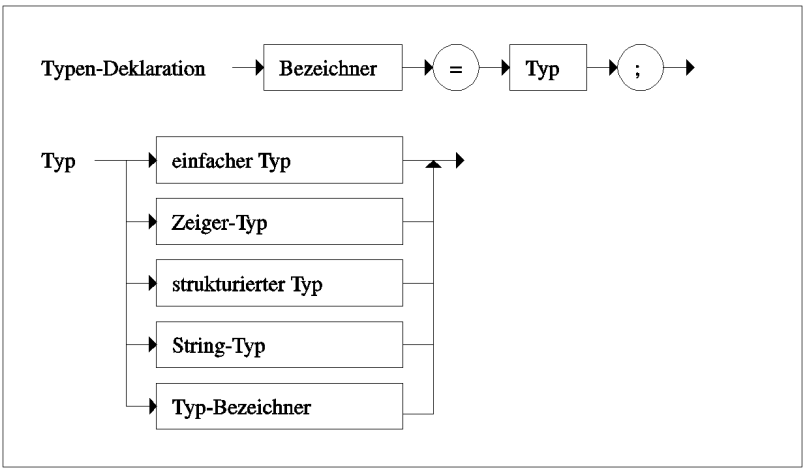


Abbildung 3.1 Syntax einer Typen-Definition

Wenn ein Bezeichner auf der linken Seite einer Typen-Definition steht, wird er für den Block, in dem seine Definition stattfindet, als Bezeichner eines Typs definiert. Der Gültigkeitsbereich eines Typen-Bezeichners erstreckt sich nicht auf die Definition selbst (d.h. die Definition kann den zukünftig definierten Typ nicht selbst enthalten). Die einzige Ausnahme von dieser Regel stellen Zeiger-Typen dar. Bezeichner werden je nach Art des Typs, für den sie stehen, in sechs Klassen unterteilt:

einfach	String
strukturiert	Zeiger
prozedural	Objekt

Einfache Typen

Einfache Typen definieren geordnete Mengen gleichartiger Werte.

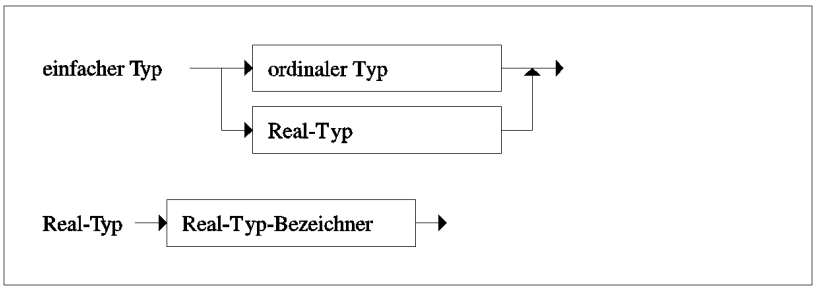


Abbildung 3.2 Syntax einfacher Typen

Der Bezeichner eines ganzzahligen Wertes steht für einen der fünf Integertypen von Turbo Pascal. Der Bezeichner eines Realwertes hat den Typ *Real*, *Single*, *Double*, *Extended* oder *Comp*. Zur Notation von Integer-Konstanten und Real-Konstanten vergleichen Sie bitte die Abschnitte »Numerische Werte« und »Zeichenketten« in Kapitel 1 »Sprachelemente und Konstanten«.

Ordinale Typen

Ordinale (d.h. abzählbare) Typen stellen eine Untermenge der einfachen Typen dar. Alle einfachen Typen mit Ausnahme der Real-Typen sind abzählbar, also ordinal. Ordinaltypen unterscheiden sich von anderen Typen durch vier Merkmale:

- Alle möglichen Werte eines gegebenen Ordinal-Typs bilden eine geordnete Menge. Jeder mögliche Wert wird einer Ordinalzahl zugeordnet, die einen ganzzahligen Wert hat. Mit Ausnahme von Integer-Werten hat der erste Wert jedes Ordinal-Typs die Ordinalzahl 0, der nächste die Ordinalzahl 1 usw. Jeder mögliche Wert eines Ordinal-Typs läßt sich auf diese Weise zuordnen. In jedem Ordinal-Typ hat jeder Wert außer dem ersten einen Vorgänger und jeder außer dem letzten einen Nachfolger, die Reihenfolge richtet sich nach der Anordnung des Typs.

- Die Standard-Funktion *Ord* kann auf jeden ordinalen Wert angewendet werden und liefert die Ordinalzahl des betreffenden Wertes zurück.

- Die Standard-Funktion *Pred* kann auf alle Ordinal-Typen angewendet werden und liefert den Vorgänger des betreffenden Wertes zurück. Der Vorgänger des ersten Wertes eines Ordinal-Typs ist nicht definiert.

■ Die Standard-Funktion *Succ* kann auf alle Ordinal-Typen angewendet werden und liefert den Nachfolger dieses Wertes zurück. Diese Funktion verhält sich ähnlich wie *Pred*: Wenn sie auf den letzten Wert eines Ordinal-Typs angewendet wird, erzeugt auch sie einen Fehler.

Die Syntax eines Ordinal-Typs läßt sich folgendermaßen darstellen:

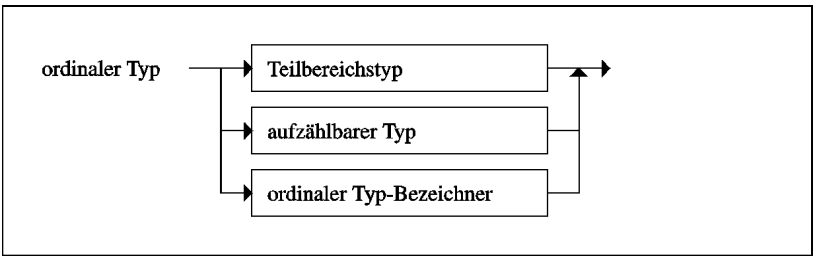


Abbildung 3.3 Syntax eines Ordinal-Typen

Turbo Pascal bietet sieben vordefinierte Ordinal-Typen: *ShortInt, Integer, LongInt, Byte, Word, Boolean* und *Char*. Zusätzlich sind zwei weitere Gruppen benutzerdefinierter Ordinal-Typen verfügbar: aufzählbare und Teilbereichstypen.

Der Typ Integer

Integerwerte sind eine Teilmenge der ganzen Zahlen. In Turbo Pascal sind fünf verschiedene Typen vordefiniert, die jeweils unterschiedliche Wertebereiche haben:

Tabelle 3.1 Vordefinierte Integertypen und Wertebereiche

Typ	Bereich		Format
ShortInt	-128	.. 127	8 Bit mit Vorzeichen
Integer	-32768	.. 32767	16 Bit mit Vorzeichen
LongInt	-2147483648	.. 2147483647	32 Bit mit Vorzeichen
Byte	0	.. 255	8 Bit vorzeichenlos
Word	0	.. 65535	16 Bit vorzeichenlos

Arithmetische Operationen mit Integertypen erzeugen Resultate mit 8, 16 oder 32 Bit, wobei die folgenden Regeln gelten:

■ Zur Speicherung von Konstanten wird der »kleinste« Typ benutzt, dessen Wertebereich den angegebenen Wert beinhaltet.

■ Verknüpfungen zweier Integerwerte über einen binären Operator (d.h. Addition, Multiplikation usw.) erzwingen eine automatische Konvertierung beider Operanden in ein *gemeinsames Format*. Als gemeinsames Format zweier Operanden ist jeweils der »kleinste« Standardtyp definiert, dessen Wertebereich beide Operanden beinhaltet – das gemeinsame Format von *Byte* und *Integer* ist also

Integer, das gemeinsame Format von *LongInt* und *Integer* ist *LongInt*. Die Operation wird mit dem gemeinsamen Format ausgeführt, das Ergebnis hat folglich ebenfalls dieses Format.

- Ausdrücke auf der rechten Seite einer Zuweisung werden unabhängig vom Format der Variablen auf der linken Seite durchgeführt.

- Werte, die ein einzelnes Byte Speicherplatz belegen (also *Char*, *Byte*, *ShortInt* und Aufzählungstypen bzw. entsprechende Teilbereiche) werden vor der Ausführung einer Operation vorzeichenrichtig auf 16 Bit erweitert und sind damit zu *Integer*- bzw. *Word*-Operanden kompatibel.

Ein Integerwert kann durch eine explizite Typ-Umwandlung jederzeit in ein anderes Integerformat überführt werden (siehe Kapitel 4 »Variablen« und 6 »Ausdrücke«).

Boolesche Datentypen

Insgesamt sind drei boolesche Datentypen definiert: *Boolean, WordBool* und *LongBool*. Diese Typen können ausschließlich einen der vordefinierten Werte *False* oder *True* annehmen. Da boolesche Werte als aufzählbarer Typ definiert sind, gelten folgende Beziehungen:

- *False* < *True*
- *Ord*(*False*) = 0
- *Ord*(*True*) = 1
- *Succ*(*False*) = *True*
- *Pred*(*True*) = *False*

- Ein Variable des Typs *Boolean* belegt ein Byte, eine solche des Typs *WordBool* zwei Bytes, eine *LongBool*-Variable vier Bytes. Wo immer möglich, sollte der Typ *Boolean* verwendet werden, der den geringsten Speicherplatz belegt. Die Typen *WordBool* und *LongBool* wurden aus Gründen der Kompatibiltät zu MS-Windowsw definiert.

- Aus disen Gründen heraus können boolesche Typen auch andere ordinale Werte als nur 0 und 1 annehmen. Ein boolesche Ausdruck wird daher dann als *False* angesehen, wenn sein ordinaler Wert 0 ist, dagegen als *True*, wenn sein ordinaler Wert ungleich 0 ist. Die booleschen Operatoren **not**, **and**, **or** und **xor** überprüfen, ob ein Wert von 0 (*False*) oder ungleich 0 (*True*) vorliegt, geben aber immer einen ordinalen Wert von 0 oder 1 zurück.

Der Typ Char

Die Grundmenge dieses Typs besteht aus Zeichen, deren Reihenfolge sich aus der Anordnung des IBM-Zeichensatzes bzw. der dazugehörigen ASCII-Codes ergibt. Der Aufruf der Funktion *Ord(Ch)*, bei dem *Ch* für einen Wert vom Typ *Char* steht, liefert die Ordinalzahl des Zeichens in *Ch* zurück.

Eine Stringkonstante, die nur aus einem Zeichen besteht, ist äquivalent zu einer Konstanten des Typs *Char*. Jeder Wert vom Typ *Char* kann über die Standardfunktion *Chr* erzeugt werden.

Aufzählbare Typen

Aufzählbare Typen definieren durch Aufzählung der Bezeichner geordnete Mengen. Die einzelnen Elemente erhalten Ordinalzahlen in der Reihenfolge ihrer Aufzählung: der erste Bezeichner erhält die Ordinalzahl 0, der zweite die Ordinalzahl 1 usw.

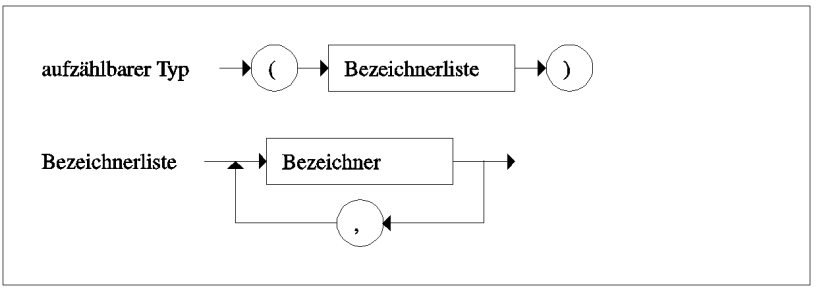


Abbildung 3.4 Syntax aufz

Wenn ein Bezeichner in der Bezeichnerliste eines aufzählbaren Typs erscheint, wird er in dem Block als Konstante deklariert, in dem auch der aufzählbare Typ deklariert wurde. Ein Beispiel für einen aufzählbaren Typ:

```
Karten = (Kreuz, Pik, Herz, Karo);
```

Durch diese Deklaration werden *Kreuz, Pik*, *Herz* und *Karo* zu Konstanten vom Typ *Karte*. Wenn die Funktion *Ord* auf einen aufzählbaren Wert angewendet wird, liefert sie einen Integer zurück, der die Position des Wertes im Verhältnis zu den anderen Werten desselben aufzählbaren Typs bezeichnet. *Ord(Kreuz)* liefert im vorangegangenen Beispiel also die Ordinalzahl 0 zurück, *Ord(Pik)* die Ordinalzahl 1 usw.

Der Teilbereichstyp

Dieser Typ umfaßt einen ordinalen Wertebereich, der in diesem Zusammenhang auch als Grundmenge bezeichnet wird. Die Definition des Teilbereichstyps legt den jeweils niedrigsten und höchsten Wert dieses Bereichs fest. Die Syntax der Definition ist:

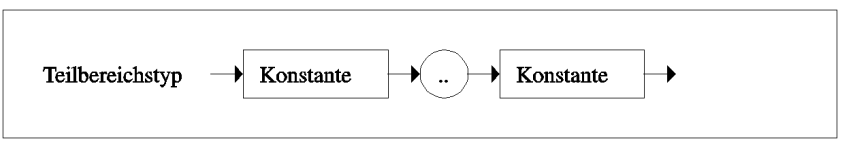


Abbildung 3.5 Syntax von Teilbereichstypen

Hier müssen beide Konstanten denselben Ordinaltyp haben. Bei einem Teilbereichstyp der Art *a..b* muß *a* kleiner oder gleich *b* sein.

Beispiele für Teilbereichstypen sind:

```
0..99
-128..127
'a'..'z'
Kreuz..Herz   {nach Definition des Typs Karte - s.o.}
```

Jede Variable eines Teilbereichstyps übernimmt die Eigenschaften der Grundmenge – ihr Wert muß sich allerdings zur Laufzeit in den durch den Teilbereich gesetzten Grenzen bewegen.

Die Sprachdefinition von Pascal legt fest, daß eine mit einer runden Klammer eingeleitete Wertefolge wie in

```
type
  Colors = (Red, Green, Blue);
```

einen Aufzählungstyp definiert, und stellt den Compiler damit vor ein Problem, wenn ein Teilbereich über einen Konstanten-*Ausdruck* deklariert werden soll. Bei einer Angabe wie

```
const
  RStart = 10;
  REnd   = 30;

type
  R_Scale = (REnd - RStart) * 2 .. (REnd + RStart) * 2;
```

ist die Absicht des Programmierers offensichtlich: es sollte ein Variablentyp definiert werden, der Werte von 40..80 speichern kann. Da sich der Compiler aber streng an die Sprachdefinition hält, erwartet er nach der öffnenden Klammer eine Folge von Wertangaben für einen Aufzählungstyp – und moniert folglich die beiden Punkte der Bereichsangabe als »unzulässig«.

Die Lösung ist recht einfach: Sie müssen lediglich dafür sorgen, daß der Ausdruck rechts des Gleichheitszeichens nicht mit einer Klammer *beginnt*:

```
type
  R_Scale = 2 * (REnd - RStart) .. (REnd + RStart) * 2;
```

Eine weitere Möglichkeit wäre, den ersten Ausdruck einer nur zu diesem Zweck definierten Konstanten zuzuweisen und diese Konstante dann für die Bereichsangabe zu verwenden, also:

```
const
  ...
  R_ScaleStart = (REnd - RStart) * 2;

type
  R_Scale = R_ScaleStart .. (REnd + RStart) * 2;
```

Was dieses Beispiel ebenfalls verdeutlicht: das »Klammerproblem« ist auf Typdefinitionen beschränkt – beim Festlegen von Konstanten-Werten macht es sich nicht bemerkbar.

Real-zahlen

Real-Typen sind eine Untermenge der reellen Zahlen und werden mit einer festgelegten Anzahl von Ziffern in Gleitkomma-Schreibweise dargestellt.

Ein Realwert *n* setzt sich normalerweise aus drei Komponenten zusammen: *m*, *b* und *e*. Hier gilt $m*b^e = n$, wobei *b* immer den Wert 2 erhält und *m* sowie *e* ganzzahlige Werte im Bereich Real sind. Außerdem bestimmen die Werte *m* und *e* sowohl die Genauigkeit als auch den Bereich des Real-Typs.

Turbo Pascal kennt fünf verschiedene Arten von Real-Typen: *Real*, *Single*, *Double*, *Extended* und *Comp*. Sämtliche Typen außer *Real* sind nur dann verfügbar, wenn mit einem numerischen Coprozessor bzw. dem Emulator gearbeitet wird (siehe nächster Abschnitt). Die einzelnen Real-Typen unterscheiden sich voneinander sowohl durch ihren Wertebereich als auch in der Genauigkeit mit ihnen durchgeführter Operationen:

Tabelle 3.2 Bereich und Rechengenauigkeit von Real-Typen

Typ	Bereich		Genauigkeit
Real	$2.9*10^{-39}$	bis $1.7*10^{38}$	11-12 Stellen
Single	$1.5*10^{-45}$	bis $3.4*10^{38}$	7-8 Stellen
Double	$5.0*10^{-324}$	bis $1.7*10^{308}$	15-16 Stellen
Extended	$3.4*10^{-4932}$	bis $1.1*10^{4932}$	19-20 Stellen

Der Typ *Comp* umfaßt nur ganzzahlige Werte im Bereich von -2^{63} bis $2^{63}-1$, was ungefähr dem Bereich von -9.2 * 1018 bis 9.2 * 10^{18} entspricht.

Turbo Pascal unterstützt zwei Arten der Code-Erzeugung für die Ausführung mathematischer Operationen mit Realzahlen, zwischen denen über den Compiler-Befehl `$N` gewählt werden kann: *Software* (Berechnung über Bibliotheksroutinen) und *8087/80287* (Berechnung über einen numerischen Coprozessor bzw. Emulation eines Coprozessors).

Fließkomma-Operationen über Bibliotheksroutinen

Der Schalter `$N` ist als Standardvorgabe auf `$N-` gesetzt, d.h. Fließkomma-Operationen finden über Aufrufe der Laufzeitbibliothek statt. Aus Gründen der Arbeitsgeschwindigkeit und des Code-Umfangs sind bei dieser Vorgabe nur Operationen mit dem Format *Real* möglich – jeder Versuch der Compilierung eines Programms, das Berechnungen mit anderen Real-Formaten enthält, endet mit einer entsprechenden Fehlermeldung.

Fließkomma-Operationen mit einem numerischen Coprozessor

Nach der Aktivierung des Schalters `$N` (mit dem Compiler-Befehl `{$N+}`) erzeugt der Compiler Code zur direkten Ansteuerung eines numerischen Coprozessors (8087, 80287 oder 80387). In diesem Modus sind sämtliche in Tabelle 3.2 aufgeführten Real-Formate für Berechnungen verfügbar.

Der Typ String

Ein String-Typ ist eine Zeichenfolge mit dynamischer Länge, dem ein Speicherbereich konstanter Größe (1-255 Zeichen) zugewiesen ist. Die tatsächliche Länge des Strings zur Laufzeit des Programms wird dynamisch verwaltet. Wenn ein String-Typ ohne Größenzuweisung deklariert wird, bekommt er die Standardgröße 255 zugewiesen. Die Funktion *Length* liefert die aktuelle Länge eines Strings zurück.

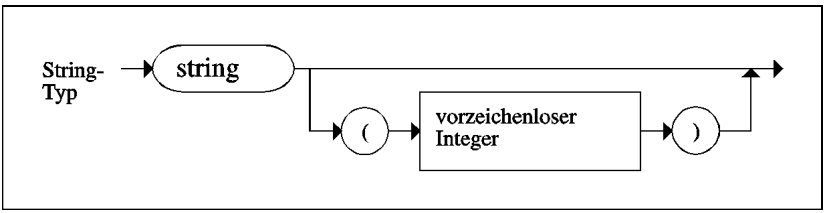


Abbildung 3.6 Syntax von String-Typen

Das Verhältnis zweier Stringwerte ergibt sich durch ihren Inhalt und durch ihre Länge: Zeichen auf gleicher Position innerhalb zweier Strings werden nach ihrem ASCII-Code bewertet ('A' ist also kleiner als 'B', 'AA' ist kleiner als 'AB', 'Z' ist kleiner(!) als 'a'. Wenn zwei Strings bis zur Position *n* dieselben Zeichen enthalten, wird der kürzere von beiden als der kleinere bewertet ('Z' ist kleiner als 'Zs'). Ein *Nullstring*, d.h. ein String ohne Inhalt, stellt den kleinstmöglichen Wert dar.

In Kapitel 4 »Variablen« ist im Abschnitt »Arrays, Strings und Indizierungen« beschrieben, wie auf einzelne Zeichen eines Strings als Komponenten eines Arrays zugegriffen werden kann. Operatoren für String-Typen werden in Kapitel 6 »Ausdrücke« in den Abschnitten »String-Operatoren« und »Relationale Operatoren« erläutert; Erklärungen zu Stringprozeduren und -funktionen finden Sie im Kapitel 24 »Verzeichnis der Routinen«.

Strukturierte Typen

Ein strukturierter Typ, der durch die Art seiner Strukturierung und durch seinen Komponenten-Typ bzw. seine Komponenten-Typen gekennzeichnet ist, enthält mehr als einen Wert. Wenn ein Komponenten-Typ strukturiert ist, hat der resultierende strukturierte Typ mehr als eine Strukturebene. Ein strukturierter Typ kann beliebig viele Strukturebenen haben.

Das reservierte Wort **packed** bei der Definition einer Datenstruktur weist den Compiler an, diese Daten in »gepackter« Form (d.h. mit einem möglichst geringen Verbrauch an Speicherplatz) zu speichern – und das sogar dann, wenn

dadurch die Zugriffsgeschwindigkeit auf eine Variable dieses Typs herabgesetzt wird. Da Turbo Pascal grundsätzlich mit gepackten Formaten arbeitet, wird **packed** lediglich formalsyntaktisch bei der Compilierung geprüft – eine weitere Wirkung hat es nicht.

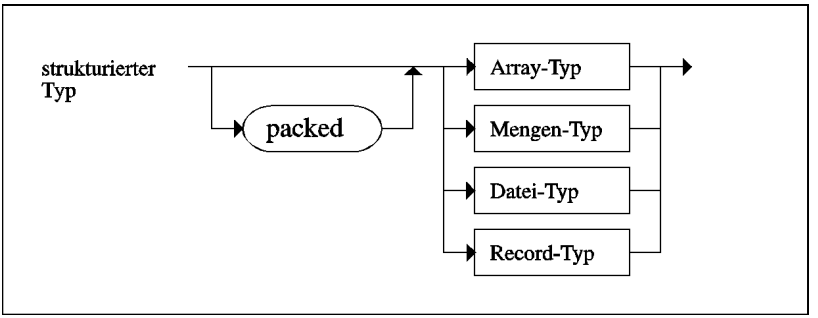


Abbildung 3.7 Syntax strukturierter Typen

Array-Typen

Arrays haben eine festgelegte Anzahl von Komponenten eines einzigen Typs, dem Komponenten-Typ. Im Syntaxdiagramm für Arrays folgt der Komponenten-Typ auf das reservierte Wort **of**.

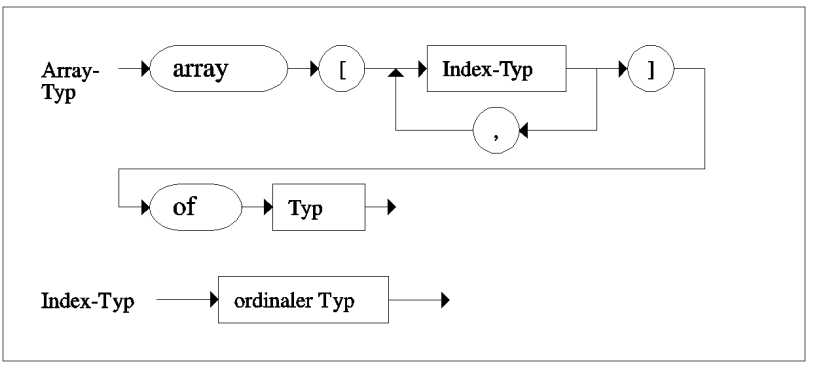


Abbildung 3.8 Syntax von Array- oder Index-Typen

Die Anzahl der Elemente jeder Dimension wird über den jeweiligen Index-Typ festgelegt, wobei alle ordinalen Grundtypen zulässig sind, außer dem Typ *Longint* und Teilbereichen von *Longint*. Die Grenzen von Teilbereichstypen müssen sich innerhalb des Wertebereichs einfacher Ordinal-Variablen bewegen: `[0..MaxInt]` stellt einen zulässigen Index-Typ dar, `[65000..75000]` dagegen nicht.

Ein Array kann in jeder Dimension über alle Werte seines Index-Typs indiziert

werden; die Maximalzahl der Elemente einer Dimension ist also mit der Anzahl der möglichen Werte des Index-Typs identisch.

Ein Beispiel für einen Array-Typ:

```
array[1..100] of Real
```

Wenn der Komponenten-Typ eines Arrays ebenfalls ein Array ist, kann man das Ergebnis entweder als ein Array **of** Array oder als ein einziges mehrdimensionales Array behandeln. Die Deklaration

```
array[Boolean] of array[1..10] of array[Anzahl] of Real
```

wird vom Compiler genauso interpretiert wie

```
array[Boolean, 1..10, Anzahl] of Real
```

Aus demselben Grund sind die beiden folgenden Deklarationen identisch:

```
packed array[1..10] of packed array[1..8] of Boolean
packed array[1..10, 1..8] of Boolean
```

Der Zugriff auf Komponenten eines Arrays geschieht durch Anfügen eines oder mehrerer Indizes in eckigen Klammern hinter dem Arraynamen (siehe Abschnitt »Arrays, Strings und Indizierungen« in Kapitel 4 »Variablen«). Wenn *m* einen kleineren Wert als *n* hat, dann wird ein Array-Typ der Form

```
packed array[m..n] of Char
```

als gepackter String bezeichnet (das reservierte Wort **packed** könnte in Turbo Pascal auch fehlen, weil sowieso »gepackt« wird). Dieser Typ hat einige Eigenschaften, über die die anderen Array-Typen nicht verfügen. Näheres dazu finden Sie im Abschnitt »Identische und kompatible Typen« in diesem Kapitel.

Record-Typen

Ein Record-Typ beinhaltet eine festgelegte Anzahl von Komponenten oder Feldern, die ihrerseits verschiedene Typen haben können. Die Definition eines Record-Typs legt den Typ und den Bezeichner jedes Feldes fest.

Der festgelegte Teil eines Record-Typen definiert eine Folge von Feldern, wobei jedes Feld einen Bezeichner und einen Typ erhält. Die Inhalte dieser Felder werden immer auf die gleiche Weise gelesen. Die folgenden Zeilen definieren einen Record-Typ des Namens *Datum* mit festgelegten Feldern:

```
type
  Datum  = record
    Jahr:  Integer;
    Monat: 1..12;
    Tag:   1..31;
  end;
```

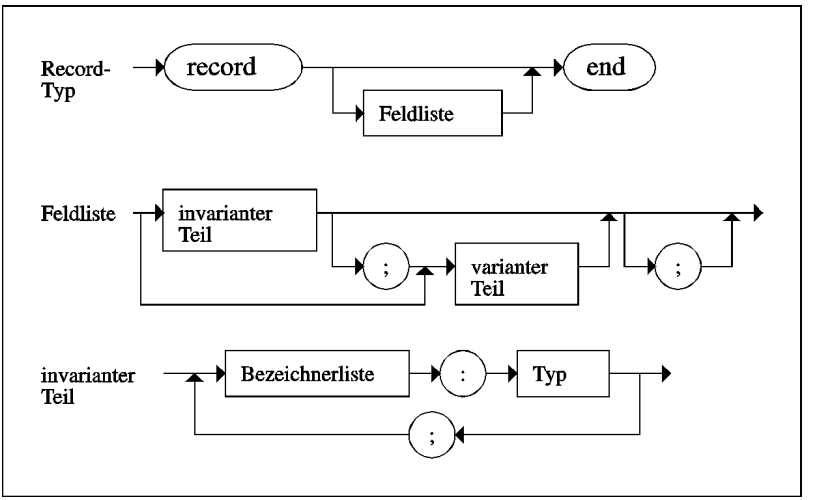


Abbildung 3.9 Syntax von Record-Typen

Der variante Teil eines Records definiert Felder, die mit Werten unterschiedlichen Typs belegt werden können: ein Feld kann beispielsweise entweder einen booleschen Wert, einen Integer oder einen String enthalten. Der Compiler weist bei varianten Feldern jeweils automatisch Speicherplatz für das größte der aufgeführten Elemente zu. Auf variante Felder kann auf mehrere Arten zuge griffen werden – je nachdem, welchen Inhalt sie zum Zeitpunkt des Zugriffs haben.

Wie das Syntaxdiagramm verdeutlicht, wird jede Variante durch mindestens eine Konstante bezeichnet. Alle Konstanten müssen eindeutig sein und einen Ordinal-Typ haben, der zum Typ des Selektorfeldes kompatibel ist. Auf Varianten wird auf dieselbe Art zugegriffen wie auf festgelegte Felder.

Durch die Definition eines weiteren (optionalen) Bezeichners im varianten Teil der Feldliste läßt sich ein *Selektorfeld-Bezeichner* festlegen, der damit zum Bezeichner eines zusätzlichen Feldes des Records – des *Selektor-Feldes* – wird. Das Programm kann den Wert dieses Feldes benutzen, um zu bestimmen, welche Variante zu einem bestimmten Zeitpunkt gerade aktiv ist. Ist kein Selektor-Feld vorhanden, muß das Programm die Art einer aktiven Variante über andere Kriterien bestimmen. Die folgenden Beispiele stellen gültige Definitionen varianter Records dar:

```
record
  x, y: Real;
     case Art: Umriss of
        Rechteck:  (Hoehe, Breite: Real);
        Dreieck:   (Groesse1, Groesse2, Winkel: Real);
        Kreis:     (Radius: Real);
end;
```

```
record
  Vorname, Nachname: string[40];
  Geburtsdatum: Datum;
  case Staatsbuerger: Boolean of
    true:   (Geburtsort: string[40];
    false:  (Geburtsland: string[20];
             Einreiseort: string[20];
             Einreisedatum: Datum;
             Ausreisedatum: Datum;
end;
```

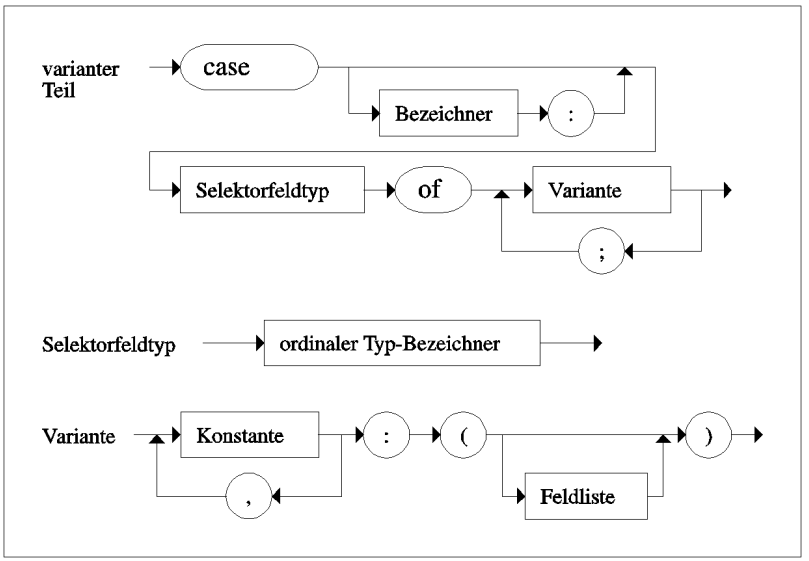


Abbildung 3.10 Syntax varianter Teile eines Records

Objekt-typen

Ein Objekttyp ist eine Struktur, die aus einer bestimmten Anzahl von Komponenten besteht. Jede dieser Komponenten ist entweder ein *Feld*, das Daten eines bestimmten Typs enthält, oder eine *Methode*, die eine Operation mit dem Objekt durchführt. Entsprechend einer Variablendeklaration, gibt man in der Deklaration eines Felds den Datentyp des Felds und eine Bezeichnung für das Feld an. Und entsprechend einer Prozedur- oder Funktionsdeklaration enthält die Deklaration einer Methode den Kopf einer Prozedur, einer Funktion, eines Konstruktors oder eines Destruktors.

Ein Objekttyp kann Komponenten von einem anderen Objekttyp *erben*. Wenn *T2* die Eigenschaften von *T1* übernimmt, dann ist *T2* ein *Nachkomme* von *T1* und *T1* ist ein Vorfahr von *T2*.

Die Vererbung ist ein transitiver Vorgang. Das bedeutet, daß *T3* auch die

Eigenschaften von *T1* erbt, wenn *T3* das Objekt *T2* und *T2* das Objekt *T1* erbt. Der *Bereich* eines Objekttyps umfaßt das Objekt selbst und alle seine Nachkommen.

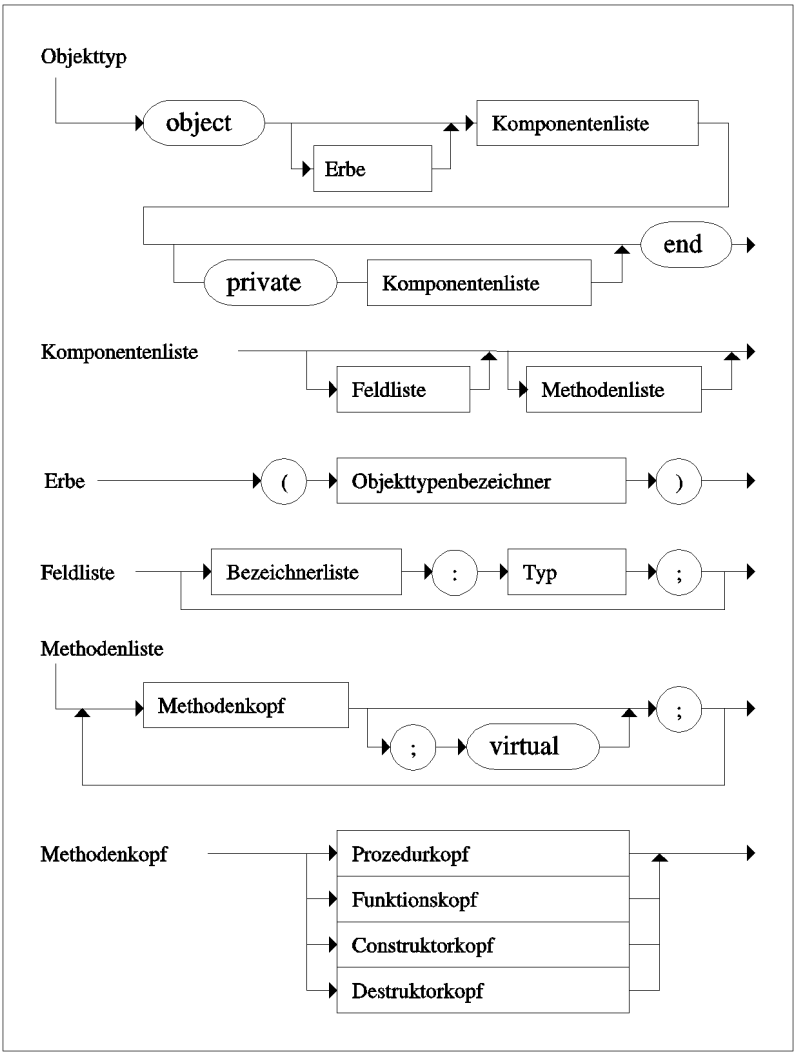


Abbildung 3.11 Syntaxerweiterungen der objektorientierten Programmierung

Es folgen einige Beispiele für die Deklaration von Objekten. Diese Deklarationen werden im weiteren Verlauf dieses Kapitels immer wieder verwendet:

```
type
  Point = object
```

```
    X, Y : Integer;
  end;

  Rect = object
    A, B : Point;
    procedure Init(XA, YA, XB, YB :Integer);
    procedure Copy(var R : Rect);
    procedure Move(DX, DY : Integer);
    procedure Grow(DX, DY : Integer);
    procedure Intersect(var R : Rect);
    procedure Union(var R : Rect);
    function Contains(P : Punkt) : Boolean;
  end;

  StringPtr = ^String;

  FieldPtr = ^Field;
  Field = object
    X, Y, Len : Integer;
    Name : StringPtr;
    constructor Copy(var F : Feld);
    constructor Init(FX, FY, FLen: Integer;FName:String);
    destructor Done; virtual;
    procedure Display; virtual;
    procedure Edit; virtual;
    function GetStr : String; virtual;
    function PutStr(S : String) : Boolean; virtual;
  end;

  StrFieldPtr = ^StrField;

  StrField = object (Field)
    Value : StringPtr;
    constructor Init(FX, FY, FLen : Integer; FName : String);
    destructor Done; virtual;
    function GetStr : String; virtual;
    function PutStr (S : String) : Boolean; virtual;
    function Get : String;
    procedure Put(S : String);
    end;

  NumFieldPtr = ^NumField;

  NumField = object (Field)
    Value, Min, Max : Longint;
    constructor Init(FX, FY, FLen : Integer; FName : String;
      FMin, FMax : Longint);
    function GetStr : String; virtual;
    function PutStr (S : String) : Boolean; virtual;
    function Get : Longint;
```

```
    procedure Put(N : Longint);
  end;

  ZipFeldPtr = ^ZipField;

  ZipField = object (Field)
    function GetStr : String; virtual;
    function PutStr(S : String) : Boolean; virtual;
  end;
```

Im Gegensatz zu anderen Datentypen kann ein Objekttyp nur in einer Typdeklaration des äußersten Gültigkeitsbereichs eines Programms oder einer Unit deklariert werden. Ein Objekttyp darf also nicht im Variablenabschnitt oder in einer Prozedur, Funktion oder Methode deklariert werden.

Unter den Komponenten eines Dateityps darf kein Objekttyp und keine Struktur mit einem Objekttyp als Komponente angegeben werden.

Der Gültigkeitsbereich eines Komponentenbezeichners umfaßt den gesamten Bereich eines Objekttyps. Darüber hinaus bezieht sich der Gültigkeitsbereich eines Komponentenbezeichners auf alle Prozedur-, Funktions-, Konstruktor- und Destruktorblöcke, in denen die Methoden des Objekttyps und seiner Nachkommen definiert sind. Aus diesem Grund muß ein Komponentenbezeichner innerhalb eines Objekttyps sowie dessen Nachkommen und in all seinen Methoden eindeutig bezeichnet sein.

Der Gültigkeitsbereich eines im **private**-Abschnitt deklarierten Bezeichners beschränkt sich auf das Modul (Programm oder Unit), das die entsprechende Objekttyp-Deklaration enthält. Das heißt, private Komponenten-Bezeichner verhalten sich wie normale öffentliche (*public*) Komponenten-Bezeichner innerhalb des Moduls, das die Objekttyp-Deklaration enthält. Außerhalb des Moduls sind die private Komponenten-Bezeichner nicht definiert. Durch Deklaration verwandter Objekttypen innerhalb desselben Moduls erhalten diese Objekttypen Zugriff zu sämtlichen Komponenten der anderen Objekttypen, ohne die **private**-Komponenten anderen Modulen sichtbar zu machen.

Die Deklaration einer Methode innerhalb eines Objekttyps entspricht einer **forward**-Deklaration dieser Methode. Die Methode muß also irgendwann hinter der Deklaration des Objekttyps innerhalb desselben Gültigkeitsbereichs wie die Deklaration des Objekttyps in Form einer Definition *implementiert* werden.

Wenn eine spezifische Unterscheidung erforderlich ist, muß der vollständige Methodenbezeichner angegeben werden. Dieser setzt sich aus dem Bezeichner des Objekttyps, einem dahinter stehenden Punkt (.) und dem Bezeichner der Methode zusammen. Wie bei jedem anderen Bezeichner kann vor dem vollständigen Methodenbezeichner noch ein Unit-Bezeichner und ein weiterer Punkt vorangestellt werden, falls dies erforderlich ist.

Innerhalb der Deklaration eines Objekttyps kann man in der Kopfzeile einer Methode die Parameter des deklarierten Objekttyps angeben, auch wenn die Deklaration noch nicht abgeschlossen ist. Dies wurde in den Methoden *Copy*, *Intersect* und *Union* des Typs *Rect* im vorherigen Beispiel veranschaulicht.

Methoden sind in der Standardeinstellung statisch, können aber mit Ausnahme der Konstruktorfunktionen durch die Anweisung **virtual** in der Deklaration der Methode als *virtuell* deklariert werden. Der Compiler löst die Aufrufe

von statischen Methoden zur Compilationszeit auf, während die virtuellen Methoden zur Laufzeit ermittelt werden. Letzteres wird auch als *dynamische Bindung* bezeichnet.

Wenn in einem Objekttyp virtuelle Methoden deklariert oder vererbt werden, müssen die Variablen dieses Typs mit einem Aufruf eines Konstruktors *initialisiert* werden, bevor die virtuelle Methode aufgerufen werden kann. In jedem Objekttyp, in dem virtuelle Methoden deklariert oder vererbt werden, muß mindestens ein Konstruktor deklariert oder vererbt werden.

Ein Objekttyp kann jede der Methoden, die von einem Vorfahren vererbt werden, *ersetzen* (neu definieren). Wenn bei der Deklaration einer Methode in einem Nachkommen dieselbe Bezeichnung verwendet wird wie bei einem Vorfahren, dann ersetzt die Deklaration im Nachkommen die Deklaration des Vorfahren. Der Gültigkeitsbereich der neuen Methode umfaßt den Bereich, in dem der Nachkomme eingeführt wurde und erstreckt sich auch auf weitere Nachkommen, es sei denn, diese ersetzen ihrerseits die betreffende Methode.

In der Neudefinition einer statischen Methode kann man die Kopfzeile der Methode beliebig angeben. Im Gegensatz dazu muß die Neudefinition einer virtuellen Methode in Bezug auf Reihenfolge, Typ und Bezeichnung der Parameter sowie dem Typ des Funktionsergebnisses, falls eine Funktion vorliegt, exakt übereinstimmen. Darüber hinaus muß in der Neudefinition wieder die Anweisung **virtual** stehen.

Turbo Pascal für Windows führt eine weitere Klasse von Methoden ein, die *dynamischen* Methoden, die sich von den oben dargestellten virtuellen Methoden nur in der Art unterscheiden, wie der Aufruf einer solchen Methode zur Laufzeit des Programms aufgelöst wird.

Die Deklaration einer dynamischen Methode entspricht weitgehend der einer virtuellen Methode, mit der Ausnahme, daß dem reservierten Wort **virtuell** die Angabe eines Indexwertes folgen muß. Der Wert dieser Integer-Konstante muß im Bereich von 1..65535 liegen und eindeutig sein, darf also nicht für mehrere Methoden eines Objekttyps oder seiner Nachkommen vergeben werden. Beispiel:

```
procedure FileOpen(var Msg: TMessage); virtual 100;
```

Überschreibt ein Nachkomme eine dynamische Methode, muß die Deklaration der überschreibenden Methode in Bezug auf Reihenfolge, Typ und Namen der Parameter der Deklaration der überschriebenen Methode entsprechen. Auch sie muß das reservierte Wort **virtual** angeben, dem der gleiche Indexwert folgt, der schon der überschriebenen Methode des Vorfahren zugeordnet wurde. Kapitel 17 »Objekte« geht auf die Details näher ein.

Ein Objekt wird *eingesetzt*, in einer »Instanz« erzeugt, indem man eine Variable oder eine typisierte Konstante eines Objekttyps deklariert oder die Standardprozedur *New* auf eine Zeigervariable eines Objekttyps anwendet. Das Ergebnis wird als *Instanz* des Objekttyps bezeichnet.

```
var
  F : Field;
  Z : ZipField;
```

```
  FP : FieldPtr;
  ZP : ZipFieldPtr;
```

In diesen Variablendeklarationen ist *F* eine Instanz von *Field*, *Z* eine Instanz von *ZipField*. In gleicher Weise zeigen *FP* und *ZP* nach dem Aufruf von *New* mit *FP* und *ZP* als Parameter auf Instanzen von *Field* und *ZipFeld*.

Wenn ein Objekttyp virtuelle Methoden enthält, dann müssen die Instanzen dieses Objekttyps mit Hilfe eines Konstruktor-Aufrufs initialisiert werden, bevor eine der virtuellen Methoden aufgerufen werden kann. Hierzu ein Beispiel:

```
var
  S: StrField;

begin
  S.Init(1, 1, 25, 'Vorname');
  S.Put('Frank');
  S.Display;
  ...
  S.Done;
end;
```

Würde *S.Init* nicht aufgerufen, führte der Aufruf von *S.Display* zu einer undefinierten Adresse. Der Aufruf *S.Put* jedoch ist eindeutig, da *Put* eine statische Methode von *StrField* ist.

Hinweis: Die Zuweisung zu der Instanz eines Objekttyps initialisiert die Instanz *nicht*.

Natürlich müssen auch solche Instanzen initialisiert werden, die Bestandteil eines strukturierten Typs sind. Zum Beispiel:

```
var
  Comment: array[1..5] of StrField;
  i: Integer;

begin
  for i:= 1 to 5 do Comment[i].Init(1, i+10, 40, 'Kommentar');
  ...
  for i:= 1 to 5 do Comment[i].Done;
end;
```

Bei dynamischen Instanzen ist in den meisten Fällen die Initialisierung mit einer Speicherplatzreservierung verbunden. Beim Aufräumen wird gleichzeitig der reservierte Speicherplatz freigegeben. Für die Heap-Verwaltung wird gewöhnlich, wie im folgenden Beispiel, die erweiterte Syntax der Standard-Prozeduren *New* und *Dispose* gebraucht:

```
var
  SP: StrFieldPtr;

begin
  New(SP, Init(1, 1, 25, 'Vorname'));
  SP^.Put('Frank');
  SP^.Display;
```

```
  ...
  Dispose(SP, Done);
end;
```

Ein Zeiger auf einen Objekttyp kann einem Zeiger eines Vorfahren zugewiesen werden. Deshalb kann ein Zeiger auf einen Objekttyp zur Laufzeit eines Programms auf eine Instanz dieses Typs oder dessen Vorfahren zeigen.

Zum Beispiel kann man einen Zeiger des Typs *ZipFieldPtr* an Zeiger des Typs *ZipFieldPtr, NumFieldPtr* und *FieldPtr* zuweisen. Während der Programmausführung kann ein Zeiger des Typs *FieldPtr* entweder auf **nil** oder auf eine Instanz von *Field, StrField, NumField* oder *ZipField* beziehungsweise jede andere Instanz eines Nachkommens von Field zeigen.

Diese Kompatibilität bei Zeigerzuweisungen gilt auch für **var**-Parameter, die als Objekttyp deklariert sind. Zum Beispiel kann die Methode *Field.Copy* an eine Instanz von *Field, StrField, NumField, ZipFeld* oder jede andere Instanz eines Nachkommens von *Field* zugewiesen werden.

Eine Methode wird über einen Bezeichner der Form *Instanz.Methode* aufgerufen, wobei *Instanz* eine Instanz eines Objekttyps angibt und *Methode* eine Funktion des entsprechenden Objekttyps ist.

Bei den statischen Methoden bestimmt der (zur Compilationszeit) *deklarierte* Typ von *Instanz*, welche Methode aktiviert wird. Zum Beispiel aktivieren die Bezeichner *F.Init* und *FP.Init* immer *Field.Init*, da der deklarierte Typ von *F* und *FP*^ in beiden Fällen *Field* ist.

Bei den virtuellen Methoden steuert der (zur Laufzeit) *tatsächlich vorliegende* Typ die Auswahl. Zum Beispiel kann der Bezeichner *FP.Edit* die Methoden *Field.Edit, StrField.Edit, NumField.Edit* oder *ZIPField.Edit* aktivieren. Welche Methode tatsächlich aktiviert wird, hängt von der jeweiligen Instanz ab, auf die *FP* zeigt.

Im allgemeinen gibt es keine Möglichkeit, um feststellen zu können, welche Methode von einem virtuellen Methodenbezeichner aktiviert werden kann. Sie können zum Beispiel eine Routine entwickeln (beispielsweise eine Eingaberoutine für einen Maskeneditor), die *FP.Edit* aktiviert und später, ohne die Routine zu verändern, auf einen noch nicht vorhergesehenen Nachkommen von *Field* anwenden. Wenn eine Erweiterung dieser Art gewünscht wird, sollten Sie einen Objekttyp verwenden, der beliebig viele Typen an Nachkommen zuläßt, anstatt einen Record-Typ mit einer begrenzten Anzahl an Varianten einzusetzen.

Mengen-Typen

Der Wertebereich eines Mengen-Typs ist die Menge eines bestimmten Ordinal-Typs, des *Grundtyps* der Menge. Jeder mögliche Wert eines Mengen-Typs ist eine Teilmenge der möglichen Werte dieses Grundtyps.

Eine Variable des Typs *Menge* kann eine beliebige Anzahl der Elemente des Grundtyps enthalten, die Definition einer leeren Menge (die kein einziges Element enthält) ist ebenfalls möglich.

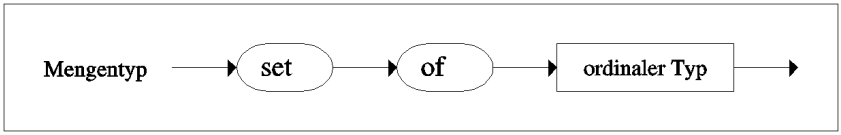


Abbildung 3.12 Syntax von Mengen-Typen

Der Grundtyp einer Menge darf nicht mehr als 256 mögliche Werte beinhalten; die niedrigste und die höchste Ordinalzahl des Typs müssen im Bereich von 0..255 liegen. Aus diesem Grund sind die Typen *ShortInt*, *Integer*, *LongInt* und *Word* nicht als Grundtyp einer Menge zulässig.

Operatoren für Mengen-Typen sind in Kapitel 6 »Ausdrücke« im Abschnitt »Mengen-Operatoren« erläutert. Der Abschnitt »Mengenkonstruktoren« desselben Kapitels erklärt, wie Mengen definiert werden. Jeder Mengen-Typ kann den Wert *[ ]* enthalten, die sogenannte *leere Menge*.

Datei-typen

Ein Dateityp besteht aus einer linearen Folge von Komponenten eines Typs, dem Komponenten-Typ. Er kann jeden beliebigen Typ haben (außer den Dateityp selber oder einen strukturierten Typ mit einer Komponente des Typs *File*). Die Anzahl der Komponenten wird durch die Deklaration des Dateityps nicht definiert.

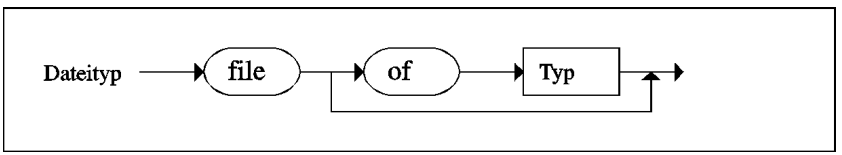


Abbildung 3.13 Syntax von Dateitypen

Wenn das reservierte Wort **of** und der Komponenten-Typ nicht angegeben sind, wird die Datei als *untypisiert* bezeichnet. Derartige Dateien werden vorwiegend für Ein- und Ausgaben auf niedriger Ebene verwendet.

Der Standard-Dateityp *Text* bezeichnet eine Datei mit zeilenweise angeordneten Zeichenfolgen. Textdateien benutzen besondere Prozeduren zur Ein- und Ausgabe, die in Kapitel 19 »Ein- und Ausgabe« erläutert werden.

Zeiger-Typen

Ein Zeiger-Typ (»Pointer-Typ«) definiert eine Menge von Werten, die auf dynamische Variablen eines festgelegten Typs (des *Grundtyps*) zeigen. Eine Variable dieses Typs enthält die Speicheradresse eines anderen Objekts (bei dem es sich in den meisten Fällen um eine dynamische erzeugte Variable handeln wird).

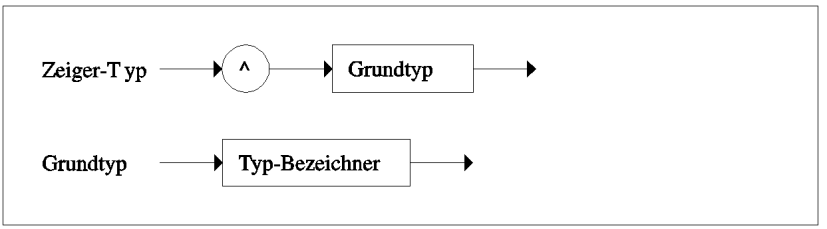


Abbildung 3.14 Syntax von Zeiger-Typen

Wenn der Grundtyp ein noch nicht deklarierter Bezeichner ist, muß er im selben Deklarationsteil wie der Zeiger-Typ vereinbart werden. Mit der Prozedur *New*, der Funktion *Ptr* oder dem Adreß-Operator @ läßt sich einer Zeiger-Variablen ein Wert zuweisen. Die Prozedur *New* weist einer dynamischen Variablen einen neuen Speicherbereich (im Heap-Bereich für Anwendungen) zu und speichert die Adresse dieser Variablen in der Zeiger-Variablen. Der Operator @ setzt die Zeiger-Variable auf die Startadresse eines Speicherbereichs, der existierende Variablen enthält – hier spielt es keine Rolle, ob diese Variablen bereits Bezeichner haben oder nicht. Die Funktion *Ptr* setzt die Zeiger-Variable auf einen bestimmten Wert, d.h. eine bestimmte Speicheradresse.

Das reservierte Wort **nil** (»not in list« = nicht in der Liste) steht für einen Zeigerwert, der *nirgendwohin* zeigt (vgl. Kapitel 4 »Variablen«).

Der Standard-Typ *Pointer* steht für einen *untypisierten* Zeiger, d.h. einen Zeiger, der auf keinen bestimmten Variablentyp zeigt. Zeigervariablen dieses Typs sind wie die Konstante **nil** mit allen anderen Zeigern kompatibel – da eine als *Pointer* deklarierte Zeigervariable *p* aber auf keinen bestimmten Variablentyp zeigt, werden Bezüge der Form *p*^ (also die Ermittlung einer Adresse) als Fehler gewertet.

Der vordefinierte Typ *PChar* bezeichnet eine Zeiger auf einem nullterminierten String und ist deklariert als

```
type PChar = ^Char;
```

Turbo Pascal für Windows unterstützt eine Reihe erweiterter Syntaxregeln, die den Umgang mit nullterminierten String über den Typ *PChar* erleichtern. Kapitel 13 »Die Unit Strings« geht auf diese Möglichkeiten näher ein.

Prozedurtypen

Eine ausführliche Besprechung der damit verbundenen Möglichkeiten finden Sie in Kapitel 8 »Prozeduren und Funktionen«.

Die Deklaration eines Prozedurtyps

legt Anzahl, Typ und Reihenfolge der Parameter sowie (für Funktionen) den Typ des Ergebnisses fest. Sie unterscheidet sich von der Deklaration einer äquivalenten Routine hauptsächlich dadurch, daß der Name nach dem reservierten Wort **procedure** bzw. **function** fehlt.

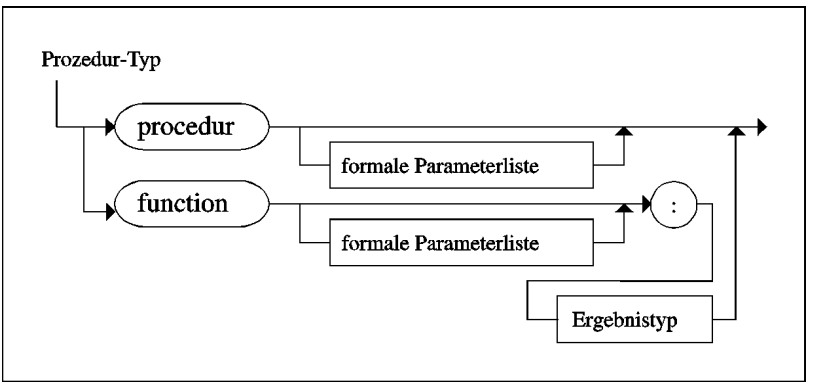


Abbildung 3.15 Syntax von Prozedur-Variablen

Einige Beispiele dazu (bei denen die angegebenen Parameternamen lediglich als Platzhalter dienen – sie müssen nicht mit den Parameternamen der Routinen übereinstimmen, die später mit diesen Typen deklariert werden):

```
type
  ProcType = procedure   { für parameterlose Prozeduren }
  SwapProc = procedure(var X,Y:Integer);
  StrProc  = procedure(s: string);
  MathFunc = function(X: Real):Real;
  Dev_Func = function(var F: Text): Integer;
  AnyFunc  = function(A, B: Real; FPtr: MathFunc);
```

Wie das letzte Beispiel zeigt, ist auch hier eine »Verschachtelung« möglich: Der Typ *AnyFunc* definiert einen Zeiger auf Funktionen, die ihrerseits einen Zeiger auf eine Funktion des Typs *MathFunc* erwarten.

Hinweis: Eine Funktion kann nur ordinale Werte, Strings, Realzahlen oder

Datenzeiger zurückliefern – Funktionen, deren Ergebnis ein Zeiger auf eine Routine ist, sind nicht definiert.

Identische und kompatible Typen

Zwei Typen können *identisch* sein, wobei diese »Gleichheit« in manchen Zusammenhängen obligatorisch ist. Für andere Operationen reicht eine *Kompatibilität* der beteiligten Typen aus (wie z.B. *Integer* und *LongInt*). Für Zuweisungen gelten spezielle Anforderungen – hier spricht man von Zuweisungs-Kompatibilität (z.B. bei der Zuweisung eines Integerwertes an eine Real-Variable).

Zwei Typen sind identisch, wenn sie entweder mit demselben Typ-Bezeichner deklariert worden sind oder ihre Definition sich aus demselben Bezeichner ableitet.

Identische Typen

Die Gleichheit zweier Typen ist nur in Prozedur- und Funktionsaufrufen und auch in diesem Fall lediglich zwischen aktuellen und formalen Variablen-Parametern gefordert.

Die beiden Typen *T1* und *T2* sind identisch, wenn eine der folgenden Bedingungen erfüllt ist: *T1* und *T2* sind derselbe Typen-Bezeichner; *T1* ist als äquivalent zu einem Typ deklariert, der mit *T2* identisch ist.

Die zweite Bedingung läuft darauf hinaus, daß *T1* nicht direkt als äquivalent zu *T2* deklariert werden muß. Aus einer Typen-Definition wie

```
T1 = Char;
T2 = T1;
T3 = Char;
T4 = T2;
```

ergeben sich *T1, T2, T3, T4* und *Char* als identische Typen. Die Vereinbarungen

```
T5 = set of Char;
T6 = set of Char;
```

vereinbaren *T5* und *T6* nicht als identische Typen, weil **set of** *Char* kein Typ-Bezeichner ist. Zwei Variablen, die in derselben Deklaration stehen, haben dagegen auch den gleichen Typ:

```
V1, V2 : set of Char;
```

Bei einer getrennten Deklaration ist das dagegen nicht der Fall:

```
V1 : set of Char;
V2 : set of Char;
```

```
V3 : Char;
V4 : Char;
```

Hier haben nur *V3* und *V4* denselben Typ, nicht aber *V1* und *V2*.

Kompatibilität von Typen

Kompatibilität zwischen zwei Typen kann beispielsweise bei Ausdrücken oder auch bei relationalen Operationen erforderlich sein. Darüber hinaus ist die Kompatibilität zwischen Typen die Voraussetzung zur Kompatibilität von Zuweisungen. Zwei Typen sind kompatibel, wenn zumindest eine der folgenden Bedingungen erfüllt ist:

- beide Typen sind gleich;
- beide Typen sind Real-Typen;
- beide Typen sind Integer-Typen;
- ein Typ ist ein Teilbereich des anderen;
- beide Typen sind Teilbereiche derselben Grundmenge;
- beide Typen sind Mengen-Typen mit kompatiblen Grundtypen;
- beide Typen sind gepackte String-Typen mit der gleichen Anzahl von Komponenten;
- ein Typ ist ein String-Typ und der andere ist ein String-Typ, ein gepackter String-Typ oder ein Char-Typ;
- ein Typ ist der vordefinierte Typ *Pointer*, der andere ein beliebiger Zeiger-Typ;
- ein Typ ist der Typ *PChar* und der andere ein Arraytyp der Form **array**[0..X] **of** Char (diese Regel gilt nur dann, wenn die erweiterte Syntax über **{$X+}** eingeschaltet wurde);
- beide Typen sind prozedurale Typen mit identischen Ergebnistypen sowie gleicher Zahl und Reihenfolge von Parametern identischen Typs.

Zuweisungs-Kompatibilität

Diese Art von Kompatibilität ist unabdingbar, wenn ein Wert zugewiesen werden soll (z.B. bei einer direkten Zuweisung mit := oder bei einer Parameterübergabe). Der Wert eines Typs *T2* kann einem Wert *T1* zugewiesen werden (d.h. *T1* := *T2*), wenn eine der folgenden Bedingungen erfüllt ist:

- *T1* und *T2* sind vom gleichen Typ; keiner von beiden ist ein Dateityp oder ein strukturierter Typ, der als Komponente einen Dateityp enthält;

- T_1 und T_2 sind kompatible Ordinal-Typen; der Wert von T_2 liegt im möglichen Wertebereich von T_1;
- T_1 und T_2 sind Real-Typen; der Wert von T_2 fällt in den möglichen Wertebereich von T_1;
- T_1 hat den Typ Real, T_2 den Typ Integer;
- T_1 und T_2 sind String-Typen;
- T_1 ist ein String-Typ, T_2 ein Char-Typ;
- T_1 und T_2 sind gepackte String-Typen;
- T_1 ist ein String-Typ, T_2 ein gepackter *(packed)* String-Typ;
- T_1 und T_2 sind kompatible Mengen-Typen; alle Elemente des Wertes von T_2 fallen in den möglichen Wertebereich von T_1;
- T_1 und T_2 sind kompatible Zeiger-Typen;
- T_1 ist vom Typ *PChar* und T_2 eine String-Konstante (diese Regel gilt nur dann, wenn die erweiterte Syntax über **{$X+}** eingeschaltet wurde);
- T_1 ist vom Typ *PChar* und T_2 ein Arraytyp der Form **array**[0..X] **of** Char (diese Regel gilt nur dann, wenn die erweiterte Syntax über **{$X+}** eingeschaltet wurde);
- T_1 und T_2 sind kompatible prozedurale Typen;
- T_1 und T_2 sind prozedurale Typen mit identischen Ergebnistypen sowie gleicher Zahl und Reihenfolge von Parametern identischen Typs;
- Ein Objekttyp T_2 ist zuweisungskompatibel mit einem Objekttyp T_1, wenn T_2 im Bereich von T_1 liegt;
- Ein Zeigertyp P_2, der auf einen Objekttyp T_2 zeigt, ist zuweisungskompatibel zu einem Zeigertyp P_2, der auf einen Objekttyp T_1 zeigt, wenn T_2 im Bereich von T_1 liegt.

Wenn keine dieser Bedingungen erfüllt, Zuweisungs-Kompatibilität aber erforderlich ist, meldet entweder der Compiler einen Fehler oder die Programmausführung bricht an der entsprechenden Stelle ab.

Der Typen-Definitionsteil

Alle Programme, Prozeduren oder Funktionen, in denen Typen definiert werden, müssen zu diesem Zweck einen Typen-Definitionsteil haben. Dieser Teil könnte beispielsweise so aussehen:

```
type
  Bereich     = Integer;
  Zahl        = Integer;
  Farbe       = (rot, gruen, blau);
```

```
CharVal     = Ord('A')..Ord('Z');    { ein berechneter Bereich }
TestIndex   = 1..100;
Testwert    = -99..99;
Testlist    = array[TestIndex] of Testwert;
TestlistPtr = ^Testlist;
Datum       = object
                Jahr : Integer;
                Monat : 1..12;
                Tag : 1..31;
                procedure SetzeTag(T, M, J: Integer);
                function ZeigeDatum: String;
              end;
Messdaten   = record
                Wann : Datum;
                Anzahl : TestIndex;
                Daten : TestlistPtr;
              end;
Messliste   = array[1..50] of Messdaten;
Name        = string[80];
Geschlecht  = (weiblich, maennlich);
Person      = ^PersonDaten;
PersonDaten = record
                Name, Vorname : Name;
                Alter : Integer;
                verheiratet : Boolean;
                Vater, Kind, Geschwister : Person;
                case s: Geschlecht of
                  maennlich : (baertig : Boolean);
                  weiblich : (schwanger :Boolean);
                end;
PersonBuffer  = array[0..SizeOf(PersonDaten)-1] of Byte;
Leute         = file of PersonDaten;
```

In diesem Beispiel sind *Bereich*, *Zahl* und *Integer* identische Typen. *TestIndex* ist zu diesen drei Typen kompatibel *und* zuweisungskompatibel – nicht aber identisch.

Hinweis: Der zulässige Bereich von *CharVal* und die Arraygröße des Typs *PersonBuffer* wurden hier mit Hilfe berechneter Konstanten definiert.

4. Variablen

Die Deklaration von Variablen

Eine Variablen-Deklaration enthält eine Liste von Bezeichnern, die ihrerseits für neue Variablen und ihren jeweiligen Typ stehen.

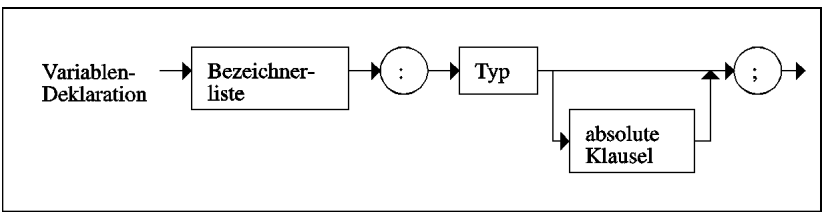


Abbildung 4.1 Syntax einer Variablen-Deklaration

Neben der Möglichkeit, einen neuen Typ innerhalb einer Variablen-Deklaration zu vereinbaren, läßt sich auch ein Typ zuweisen, dessen Typ-Bezeichner bereits vorher im Deklarationsteil desselben Blocks, in einem übergeordneten Block oder einer Unit vereinbart worden ist. Beispiele für gültige Variablen-Deklarationen sind:

```
var
  x, y, z: Real;
  i, j, k: Integer;
  Ziffer: 0..9;
  c: Farbe;
  Fertig, Fehler: Boolean;
  Operator: (plus, minus, mal);
  Hintergrund: set of Farbe;
  Hintergrundfarben: array[0..Farbe-1] of Byte;  { berechnet }
  Heute: Datum;
  Ergebnis: Messliste;
  P1, P2: Person;
  Matrix: array[1..10, 1..10] of Real;
```

Wenn ein Bezeichner in der Bezeichnerliste eines Deklarationsteils steht, gilt er innerhalb des gesamten Blocks, für den er deklariert wurde. Es kann über den ganzen Block hinweg auf diese Variable Bezug genommen werden, solange nicht in einem untergeordneten Block derselbe Bezeichner für eine andere Variable verwendet wird (Redeklaration). Eine redeklarierte Variable benutzt den Namen eines bereits existierenden Bezeichners, stellt aber ansonsten eine eigenständige Einheit dar. Der Wert der ursprünglichen Variablen wird durch eine Redeklaration nicht beeinflußt.

Variablen, die außerhalb von Prozeduren und Funktionen deklariert sind, werden als *global* bezeichnet und im Daten-Segment gespeichert. Innerhalb von Prozeduren und Funktionen deklarierte Variablen sind *lokal*, ihre Speicherung erfolgt im Stack-Segment.

Das Daten-Segment

Der letzte Schritt des Compilers besteht aus einem Verbinden des erzeugten Codes und der dazugehörigen Daten zu einem lauffähigen Programm. Bei diesem Schritt wird sämtlichen globalen Variablen des Programms sowie sämtlichen globalen Variablen aufgenommener Units ein Platz im Daten-Segment zugeordnet, das maximal 65520 Bytes umfassen kann.

Wenn Sie mehr als 65520 Bytes für globale Variablen benötigen, müssen Sie mit dynamischer Speicherverwaltung arbeiten, d.h. den Platz für größere Variablenkonstrukte über Zeiger während der Laufzeit des Programms belegen. Der Abschnitt »Zeiger und dynamische Variablen« weiter hinten in diesem Kapitel erläutert die Details.

Das Stack-Segment

Die Größe dieses Segments wird über den Compiler-Befehl `$M` festgelegt und kann zwischen 1024 und 65520 Bytes betragen; standardmäßig gibt Turbo Pascal für Windows 8192 Bytes vor.

Jede Prozedur oder Funktion belegt bei ihrer Aktivierung (d.h. ihrem Aufruf) Platz für übergebene Parameter und lokale Variablen im Stack-Segment, der beim Rücksprung zur Aufrufstelle automatisch wieder freigegeben wird.

Der Gesamtbedarf an Speicherplatz im Stack-Segment ergibt sich aus der Summe des Einzelbedarfs sämtlicher Prozeduren und Funktionen, die zu einem gegebenen Zeitpunkt gleichzeitig aktiv sein können – er darf die vorgegebene Größe des Stack-Segments nicht überschreiten. Turbo Pascal enthält aus diesem Grund den Compiler-Schalter `$S`, der standardmäßig aktiv gesetzt ist (`{$S+}`): vor jedem Aufruf einer Prozedur oder Funktion wird geprüft, ob noch ausreichend Platz im Stack-Segment vorhanden ist. Mit dem Befehl `$S-` läßt sich diese automatische Überprüfung abschalten (was erst geschehen sollte, wenn ein Programm komplett fehlerbereinigt ist – Stack-Überläufe haben meist einen vollständigen Absturz des Systems zur Folge).

Absolute Variablen

In Turbo Pascal kann die Speicheradresse einer Variablen über den Zusatz **absolute** auch direkt (d.h. über die Angabe eines fixen Wertes) festgelegt werden:

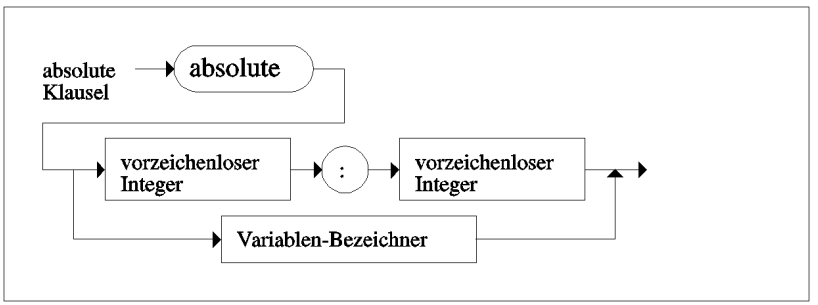


Abbildung 4.2 Syntax einer absoluten Variablendeklaration

Im Gegensatz zu allen anderen Deklarationen kann über den Zusatz **absolute** jeweils nur eine Variable vereinbart werden. Die erste Form einer Deklaration mit **absolute** gibt die Startadresse der Variablen in der Form *Segment:Offset* an:

```
var
  CrtModus:   Byte absolute $0040:$0049;
```

Die erste Konstante gibt die Segmentadresse an, die zweite den *Offset* (d.h. die Adresse innerhalb dieses Segments). Beide Konstanten müssen im Bereich von $0000 bis $FFFF (dezimal: 0..65535) liegen.

Die zweite Form deklariert eine Variable, die denselben Speicherplatz wie eine andere Variable belegt, und stellt praktisch eine Alternative zu varianten Teilen eines Records dar:

```
var
  Str:    string[32];
  StrLen: Byte absolute Str;
```

In diesem Beispiel wird festgelegt, daß die Variable *StrLen* auf derselben Adresse wie das erste Byte der Variablen *Str* gespeichert wird. Da das erste Byte eines Strings die Länge enthält, kann die momentane Länge von *Str* hier also auch über *StrLen* ermittelt werden.

Der Bezug auf Variablen

Der Bezug auf eine Variable bezeichnet:

- eine Variable;
- eine Komponente eines strukturierten Typs oder einer Variablen vom Typ *String*;
- eine dynamische Variable, auf die eine Zeiger-Variable zeigt.

Das folgende Syntaxdiagramm verdeutlicht die Referenzmöglichkeit auf eine Variable:

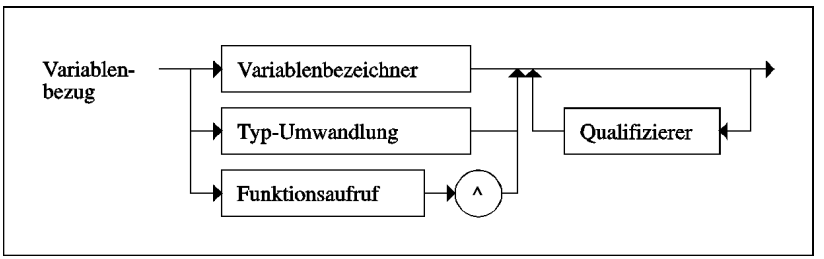


Abbildung 4.3 Syntax eines Variablenbezugs

Hinweis: Die Syntax eines Variablenbezugs ermöglicht den Aufruf von Funktionen, die als Ergebnis einen Zeiger zurückliefern. Dieses »Zeiger-Ergebnis« wird von Turbo Pascal automatisch in die tatsächliche Adresse der Variablen umgewandelt (»de-referencing«).

Die Festlegung eines Variablenbezugs

Der Bezug auf eine Variable geschieht über einen Variablenbezeichner, dem optional ein oder mehrere Werte zur Qualifizierung folgen können. Diese Werte (»Qualifizierer«) legen fest, in welchem Zusammenhang auf eine Variable Bezug genommen wird.

Ein Array-Bezeichner ohne zusätzliche Angaben bezieht sich auf das gesamte Array. Ein Beispiel:

```
Ergebnisse
```

Ein Array-Bezeichner mit einem darauffolgenden Index steht für eine bestimmte Komponente des Arrays:

```
Ergebnisse[Index + 1]
```

Ist diese Komponente ein Record oder ein Objekt, kann dem Index ein

Feldbezeichner angefügt werden. Der Variablenzugriff bezeichnet in diesem Fall ein bestimmtes Feld in einer bestimmten Komponente des Arrays.

```
Ergebnisse[Index + 1].Daten
```

Auf den Bezeichner eines Feldes, das einen Zeigerwert enthält, kann das Zeigersymbol ^ folgen. Dieses Symbol unterscheidet zwischen dem Feld selbst und der dynamischen Variablen, auf die sein Inhalt zeigt.

```
Ergebnisse[Index + 1].Daten^
```

Wenn die Variable, auf die ein Zeiger zeigt, ein Array ist, können zusätzliche »Indexe« hinzugefügt werden, die für einzelne Komponenten dieses Arrays stehen.

```
Ergebnisse[Index + 1].Daten^[J]
```

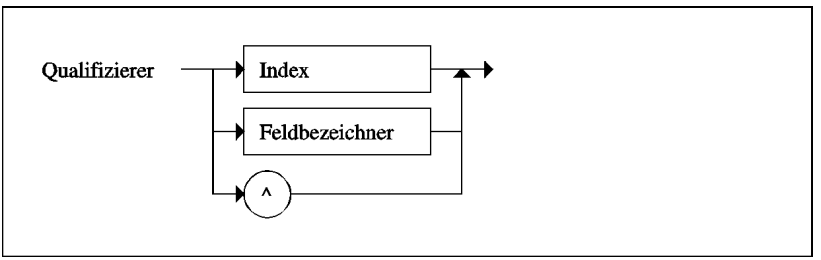


Abbildung 4.4 Syntax von Werten zur Qualifizierung von Variablenbezeichnern

Arrays, Strings und Indizierungen

Der Bezug auf eine Komponente einer Array-Variablen geschieht über die Angabe eines Index nach dem Bezeichner der Array-Variablen. Der Bezug auf ein Zeichen innerhalb einer String-Variablen geschieht auf dieselbe Weise, nämlich ebenfalls über die Angabe eines Index nach dem Bezeichner der String-Variablen.

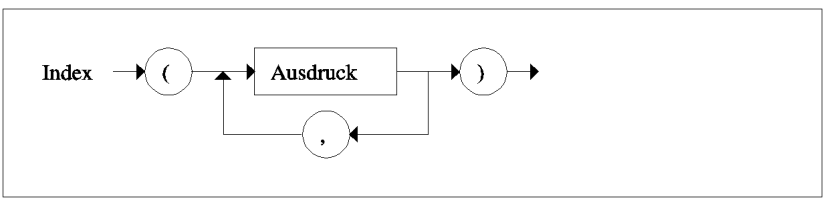


Abbildung 4.5 Syntax von Index-Ausdrücken

Die Index-Ausdrücke wählen die Komponenten aus den entsprechenden Dimensionen des Arrays aus. Die Anzahl der Index-Ausdrücke darf die der Index-Typen innerhalb der Array-Deklaration nicht überschreiten. Der Typ

jedes Ausdrucks muß zu dem des entsprechenden Index-Typen zuweisungskompatibel sein. Bei der Indizierung mehrdimensionaler Arrays können mehrere Notationsformen verwendet werden. Die folgenden beiden Variablenbezüge sind identisch:

```
Matrix[i][j]
Matrix[i,j]
```

Ein Element einer String-Variablen (d.h. ein einzelnes Zeichen) läßt sich durch einen einzigen Index-Ausdruck darstellen. Der Wert dieses Ausdrucks liegt im Bereich von 0 bis *n*, wobei *n* die deklarierte Größe des Strings darstellt. Das Ergebnis eines derartigen Bezugs hat den Typ *Char*.

Das erste Zeichen der String-Variablen (Index 0) enthält die dynamische Länge des Strings (d.h. *Length(S)* entspricht *Ord(S[0])*. Direkte Zuweisungen an das Längenattribut sind – syntaktisch gesehen – zulässig, der Compiler nimmt keine spezielle Prüfung vor.

Indizierungen außerhalb der momentanen Länge eines Strings lesen »zufällige« Werte bzw. verändern den aktuellen dynamischen Stringinhalt nicht. Zuweisungen mit Indizierungen, die außerhalb der deklarierten Maximallänge eines Strings liegen, haben dieselbe Wirkung wie unzulässige Indizierungen eines Arrays: sie zerstören andere Variablen.

Ist die erweiterte Syntax (mittels des Compiler-Befehls **{$X+}**) eingeschaltet, kann auch auf einen Wert des Typs *PChar* indiziert zugegriffen werden. Der Index ist in diesem Fall von dem Typ Word und gibt den *Offset* an, der dem Zeiger hinzuaddiert wird, bevor über ihn eine Variable des Typs Char referenziert wird.

Records und Feldbezeichner

Der Bezug auf ein Feld eines Records geschieht über einen Bezug auf die jeweilige Record-Variable, deren Bezeichner von einem Punkt und dem Feldnamen gefolgt wird.

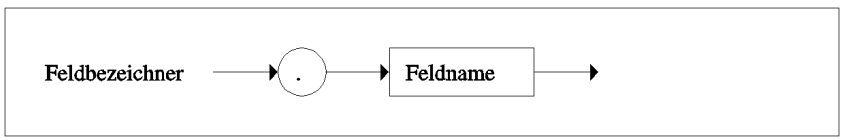


Abbildung 4.6 Syntax eines Feldbezeichners

Gültige Beispiele für Feldbezeichner sind:

```
Heute.Jahr
Messdaten[1].Zaehlung
Messdaten[1].wann.Monat
```

Anweisungen, die innerhalb eines **with**-Blocks stehen und sich auf Felder

eines mit **with** adressierten Records beziehen, müssen den Record-Bezeichner selbst nicht mehr explizit aufführen.

Bezeichner von Objektkomponenten

Das Format eines Objektkomponentenbezeichners ist dasselbe wie bei einem Feldbezeichner eines Records (siehe Kapitel 3 »Typen«). Hinter der Instanz (einer Variablenreferenz) kommt ein Punkt und ein Komponentenbezeichner. Einen Komponentenbezeichner für eine Methode nennt man Methodenbezeichner. Auf eine Instanz eines Objekttyps kann eine **with**-Anweisung angewendet werden. In diesem Fall darf man die Instanz und den Punkt beim Zugriff auf die Komponenten des Objekttyps weglassen.

Die Instanz und den Punkt kann man auch innerhalb von Methodenblöcken weglassen. Dies entspricht der Angabe von Self und einem Punkt vor der Komponentenreferenz.

Zeiger und dynamische Variablen

Ein Zeiger hat entweder den Wert **nil** oder enthält die Adresse einer (dynamischen) Variablen. Im letzteren Fall *zeigt* er auf diese Variable.

Zeigt eine Zeiger-Variable auf eine dynamische Variable, wird der Bezug über das Zeigersymbol ^ (hinter dem Bezeichner der Zeiger-Variablen) notiert. Dynamische Variablen und ihre entsprechenden Zeigerwerte werden mit der Standard-Prozedur *New* erzeugt. Der Operator @ und die Standardprozedur *Ptr* erzeugen Zeigerwerte, die in derselben Weise wie Zeiger auf dynamische Variablen behandelt werden. Der Wert **nil** steht für »not in list« (nicht in der Liste) und »zeigt« *nirgendwohin*. Ein Zugriff über einen Zeiger mit dem Wert **nil** hat einen Laufzeitfehler zur Folge. Gültige Beispiele für Bezüge auf Variablen über Zeiger sind:

```
P1^
P1^.Geschwister
Ergebnisse[1].Daten^
```

Umwandlung von Variablentypen

Der Bezug auf eine Variable eines bestimmten Typs kann in den Bezug auf eine Variable eines anderen Typs umgewandelt werden.

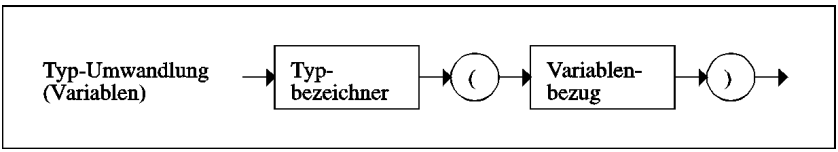


Abbildung 4.7 Syntax der Umwandlung eines Variablentyps

Wenn ein Variablenbezug umgewandelt wird, gilt dieser Bezug als Prototyp des Typen, der durch den Typ-Bezeichner festgelegt wurde. Die Größe der Variablen (also die Anzahl von Bytes, die sie im Speicher belegt) muß exakt der Größe des Typs entsprechen, für den der Typ-Bezeichner steht. Auf die Umwandlung eines Variablen-Typs kann eine beliebige Anzahl von Indizierungen folgen, soweit sie für den angegebenen Typ zulässig sind.

Einige Beispiele für die Umwandlung von Variablen-Typen:

```
type  Punkt = record
                x,y: Integer;
              end;
      Liste = array[1..2] of Integer;
var   P : Punkt;
      L : LongInt;
      N : Integer,

begin
  P := Punkt(L);
  N := Punkt(L).x;
  LongInt(P) := LongInt(P) + $00080008;
  Liste(P) [N] := 32;
end.
```

Mit diesem Prinzip läßt sich auch eine Erweiterung der Standardfunktionen *Lo* und *Hi* von *Byte*- auf *Word*-Variablen konstruieren. Nach der Definition eines Record-Typs wie

```
type  WordRec = record
                  Low, High: Word;
                end;
```

kann das Programm mit den folgenden Konstruktionen auf das nieder- bzw. höherwertige Wort einer *LongInt*-Variable *L_Var* zugreifen:

```
WordRec(L_Var).Low    { niederwertiges Wort }
WordRec(L_Var).High   { höherwertiges Wort }
```

Analoges gilt für das »Zerlegen« eines Zeigerwertes in seine beiden Anteile *Segment* und *Offset*:

```
program PointerPartDemo

type  PtrParts = record
                   Ofs, Seg: Word; { Offset auf der niedrigeren Adresse!}
                 end;
```

```
var  p: Pointer;

begin
  p := Ptr($1234, $4597);
  Writeln(PtrPart(p).Ofs);   { $4597 }
  Writeln(PtrPart(p).Seg);   { $1234 }
end.
```

Hinweis: Turbo Pascal ermöglicht zusätzlich die Typ-Umwandlung des Wertes eines Ausdrucks. Erläuterungen dazu finden Sie in Kapitel 6 »Ausdrücke«.

Außerdem unterstützt der Compiler die Typ-Umwandlung von prozeduralen Typen. Mit den Deklarationen

```
type
  Func = function(X: Integer): Integer;

var
  F : Func;
  P : Pointer;
  N : Integer;
```

können beispielsweise die folgenden Anweisungen konstruiert werden:

```
F := Func(P);   { Umwandlung von P in den Typ Func und Zuweisung }
@F := P;        { direkte Zuweisung von P an die als Pointer umgewan-
                  delte Prozedur-Variable F }
Func(P) := F;   { Zuweisung von F an die temporär umgewandelte
                  Pointer-Variable P }
P := @F;        { dito, nur daß hier F zuvor in den Typ Pointer
                  gewandelt wurde }

N := F(10);        { Aufruf einer Funktion über F und Zuweisung des
                     Funktionsergebnisses an die Variable N }
N := Func(P)(10);  { dito, nur daß der Aufruf hier über P erfolgt und
                     P zuvor in den Typ Func gewandelt wurde }
```

Hinweis: Die Anwendung auf Prozedur-Variablen stellt den einzigen Fall dar, in dem der Adreß-Operator @ auf der linken Seite einer Zuweisung erscheinen kann.

5. Typisierte Konstanten

Typisierte Konstanten sind mit initialisierten Variablen vergleichbar, d.h. mit Variablen, deren Werte beim Aufruf ihres jeweiligen Blocks festgelegt werden. Anders als bei einer untypisierten Konstanten enthält die Deklaration sowohl den Typ als auch den Wert (siehe Abschnitt »Deklaration von Konstanten« in Kapitel 1 »Sprachelemente und Konstanten«).

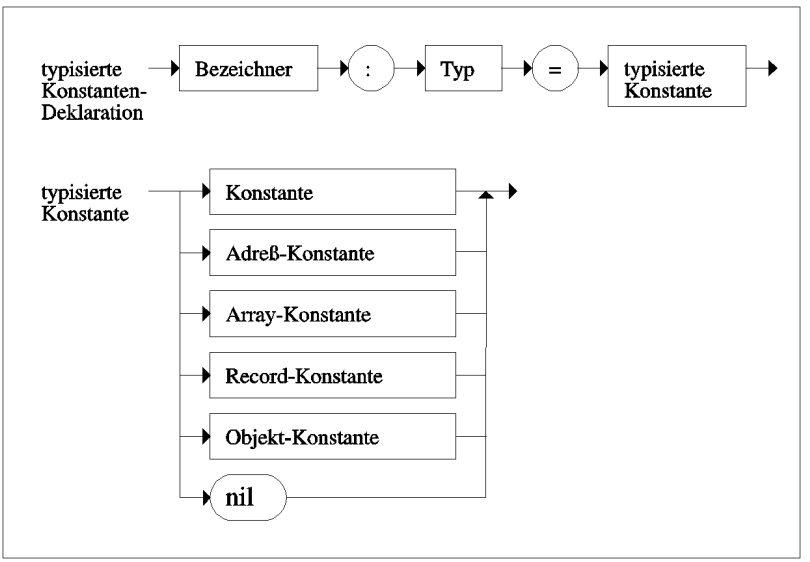


Abbildung 5.1 Syntax einer typisierten Konstantendeklaration

Typisierte Konstanten können genauso wie Variablen des entsprechenden Typs verwendet werden und dürfen auf der linken Seite einer Zuweisung erscheinen. Bitte beachten Sie, daß derartige Konstanten *einmal* initialisiert werden, nämlich beim Start des Programms. Das gilt auch für lokale Deklarationen: die Initialisierung wird also *nicht* bei jedem Aufruf der entsprechenden Prozedur oder Funktion wiederholt.

Statt mit normalen konstanten Ausdrücken können die Werte von typisierten Konstanten auch mit Hilfe von konstanten Adreß-Ausdrücken definiert

werden. Ein konstanter Adreß-Ausdruck ist ein Ausdruck, der die Adresse, das Offset oder das Segment einer globalen Variable, einer typisierten Konstante, einer Prozedur oder einer Funktion referenziert. Konstante Adreß-Ausdrücke können keine lokalen Variablen oder dynamischen Variable enthalten, weil ihre Adressen zur Compile-Zeit nicht bekannt sind.

Einfache typisierte Konstanten

Die Deklaration einfacher typisierter Konstanten enthält lediglich den Typ und einen Wert, wie die Beispiele auf der nächsten Seite zeigen:

```
const
  Maximum   : Integer = 9999;
  Faktor    : Real = -0.1;
  CtrlC     : Char = #3;
```

Wie oben schon angedeutet, kann der Wert einer typisierten Konstanten durch einen konstanten Adreß-Ausdruck definiert werden. Hierzu ein Beispiel:

```
var
  Buffer: array[0..1023] of Byte;

const
  BufferOfs: Word= Ofs(Buffer);
  BufferSeg: Word= Seg(Buffer);
```

Da eine typisierte Konstante technisch gesehen eine Variable mit einem vorgegebenen Wert darstellt, ist sie nicht mit »echten« Konstanten austauschbar. Der Versuch, sie in der Deklaration anderer Konstanten und/oder Typen zu verwenden, erzeugt eine Fehlermeldung des Compilers:

```
const
  Min  : Integer = 0;
  Max  : Integer = 99;

type
  Vecs: array[Min..Max] of Byte; {"Cannot evaluate this expression"}
```

Da *Min* und *Max* hier typisierte Konstanten sind, ist ihr Wert dem Compiler unbekannt – folglich endet die Definition des Typs *Vektor* mit einer Fehlermeldung.

Strings

Die Deklaration einer typisierten String-Konstanten enthält die Maximallänge des Strings und seinen Wert:

```
const
  Titel  : string[11] = 'Überschrift';
  Newline: string[2] = #13#10;
  StrJa  : string[3] = 'Yes';
  StrNjet: string[2] = 'No';
```

Strukturierte Typen

Die Deklaration einer typisierten strukturierten Konstanten enthält Typ und Wert jeder einzelnen Komponente. Turbo Pascal kennt Arrays, Records, Mengen und Zeiger als Konstanten dieser Art – Dateitypen sowie Arrays oder Records, die ihrerseits Dateien als Komponenten enthalten, können nicht als typisierte Konstante deklariert werden.

Arrays

Die Deklaration eines Arrays als typisierte Konstante enthält die Werte der einzelnen Array-Elemente, die dem Arraynamen in Klammern folgen und durch Kommas voneinander getrennt sind:

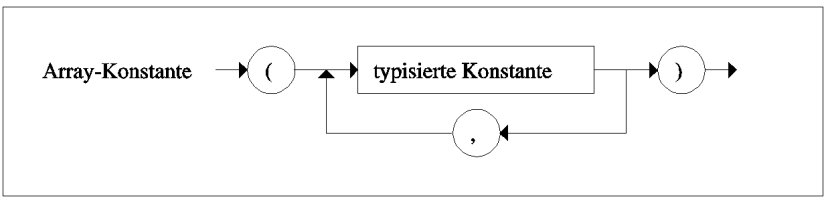


Abbildung 5.2 Syntax von Arrays als typisierte Konstanten

Ein Beispiel dazu:

```
type
  Status = (aktiv, passiv, wartend);
  StatusMap = array[Status] of string[7];

const
  StatStr: StatusMap = ('Aktiv', 'Passiv', 'Wartend');
```

Hier wird die Array-Konstante *StatStr* deklariert, über die sich beispielsweise Werte des Typs *Status* in entsprechende Strings konvertieren lassen. Die einzelnen Komponenten von *StatStr* sind:

```
StatStr[aktiv]   = 'Aktiv';
StatStr[passiv]  = 'Passiv';
StatStr[wartend] = 'Wartend';
```

Der Komponenten-Typ einer Array-Konstanten ist beliebig – nur Dateien sind nicht erlaubt. Gepackte String-Typen (d.h. Zeichen-Arrays) können entweder als einzelne Zeichen oder auch als Strings angegeben werden. Die beiden folgenden Deklarationen sind deshalb identisch:

```
const
  Digits: array[0..9] of Char = ('0', '1', '2', '3', '4',
                                 '5', '6', '7', '8', '9');

const
  Digits: array[0..9] of Char = '0123456789';
```

Ist die erweiterte Syntax (mittels des Compiler-Befehls **{$X+}**) eingeschaltet, kann ein Arraytyp der Form **array**[`0..X`] **of** Char mit einem String initialisiert werden, dessen Länge kürzer als die deklarierte Größe des Arrays ist:

```
const
  DateiName: array[0..79] of Char = 'TEST.PAS';
```

Die durch den String nicht iniatilisierten Stellen werden mit #0-Zeichen gefüllt, wodurch das Array faktisch einen nullterminierten String speichert.

Bei mehrdimensionalen Array-Konstanten werden die einzelnen Elemente jeder Dimension in einem eigenen Klammerpaar angegeben, die Klammerpaare gleicher Ebenen sind durch Kommas voneinander getrennt. Eine Deklaration wie

```
type
  Kubus = array[0..1, 0..1, 0..1] of Integer;

const
  Wuerfel : Kubus = (((0,1), (2,3)), ((4,5), (6,7)));
```

erzeugt ein Array *Wuerfel* mit den folgenden Werten:

```
Wuerfel[0,0,0] = 0
Wuerfel[0,0,1] = 1
Wuerfel[0,1,0] = 2
Wuerfel[0,1,1] = 3
Wuerfel[1,0,0] = 4
Wuerfel[1,0,1] = 5
Wuerfel[1,1,0] = 6
Wuerfel[1,1,1] = 7
```

Records

Die Deklaration eines Records als typisierte Konstante enthält Bezeichner und Wert jedes Feldes, eingeschlossen in Klammern und durch Kommas voneinander getrennt:

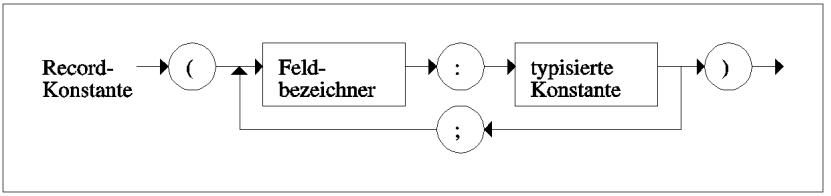


Abbildung 5.3 Syntax von Records als typisierte Konstanten

Einige Beispiele dazu:

```
type
  Punkt  = record
             x,y: Real;
           end;
  Vektor = array[0..1] of Punkt;
  Monat  = (Jan,Feb,Mar,Apr,Mai,Jun,Jul,Aug,Sep,Okt,Nov,Dez);
  Datum  = record
             Tag: 1..31; Mon: Monat; Jahr: 1900..1999;
           end;

const
  Ursprung:  Punkt = (x: 0.0; y: 0.0);
  Linie   :  Vektor = ((x: -3.1; y: 1.5), (x: 5.8; y: 3.0));
  Heute   :  Datum =(Tag: 4; Monat: Okt; Jahr: 1987);
```

Die Felder müssen in derselben Reihenfolge wie bei der Definition des Record-Typs angegeben werden. Wenn ein Record Dateitypen enthält, kann das entsprechende Feld in einer Konstanten nicht deklariert werden.

Wenn der Record variante Teile enthält, ist bei einer typisierten Konstanten nur die Angabe der aktiven Varianten möglich; enthält der Record zusätzlich ein Selektorfeld, so muß dessen Wert entsprechend angegeben sein.

Konstanten mit Objekttypen

Bei der Deklaration einer Objekttypkonstanten wird die gleiche Syntax wie bei der Deklaration von Konstanten mit Record-Typen verwendet. Für die Methoden wird (und darf auch) kein Wert angegeben werden. Zu den bereits vorher angegebenen Typdeklarationen einige Beispiele mit Objekttypkonstanten:

```
const
  ZeroPoint  : Point = (X : 0; Y : 0);
  ScreenRect : Rect =
     (A : (X : 0, Y : 0); B : (X : 80; Y : 25));
  CountField : NumField = (X : 5; Y : 20; Len : 4; Name : nil;
    Value : 0; Min : -999; Max : 999);
```

Konstanten eines Objekttyps, die virtuelle Methoden enthalten, müssen nicht mit Hilfe eines Konstruktoraufrufs initialisiert werden. Diese Initialisierung wird automatisch vom Compiler vorgenommen.

Mengen

Die Deklaration einer Menge als typisierte Konstante enthält die Elemente der Menge, eingeschlossen in eckige Klammern und durch Kommas voneinander getrennt.

Sowohl die Angabe einer leeren Menge (0 Elemente) als auch die Angabe einer Anzahl von Elementen (kleinstes Element, zwei Punkte, größtes Element) ist möglich:

Einige Beispiele dazu:

```
type
  Digits  = set of 0..9;
  Letters = set of 'A'..'Z';

const
  GeradeZiffern : Digits = [0,2,4,6,8];
  Vokale        : Letters = ['A', 'E', 'I', 'O', 'U'];
  HexZiffern    : set of '0'..'z'= ['0'..'9', 'A'..'F', 'a'..'f'];
```

Zeiger

Deklarationen von Zeigern als typisierte Konstante gebrauchen typischerweise einen konstanten Adreß-Ausdruck zur Definition des Zeiger-Wertes. Hierzu einige Beispiele:

```
type
  Richtung = (links, rechts, auf, ab);
  StringPtr = ^String;
  NodePtr = ^Node;
  Node = record
    Naechster: NodePtr;
    Symbol: StringPtr;
    Wert: Richtung;
  end;

const
  S1: String[6] = 'Ab';
  S2: String[6] = 'Auf';
  S3: String[6] = 'Rechts';
  S4: String[6] = 'Links';
  N1: Node = (Naechster: nil; Symbol: @S1; Wert: ab);
  N2: Node = (Naechster: @N1; Symbol: @S2; Wert: auf);
  N3: Node = (Naechster: @N2; Symbol: @S3; Wert: rechts);
```

```
  N4: Node = (Naechster: @N3; Symbol: @S4; Wert: links);
  RichtungsTab: NodePtr = @N4;
```

Ist die erweiterte Syntax (mittels des Compiler-Befehls **{$X+}**) eingeschaltet, kann eine typisierte Konstante vom Typ *PChar* mit einem String initialisiert werden:

```
const
  Botschaft: PChar = 'Programm beendet';
  Prompt: PChar = 'Bitte Wert eingeben: ';
  Zahlen: array[0..9] of PChar = (
    'null', 'eins', 'zwei', 'drei', 'vier',
    'fünf', 'sechs', 'sieben', 'acht', 'neun');
```

Als Ergebnis weist der Zeiger auf einen Speicherbereich mit einer nullterminierten Kopie des übergebenen Strings.

Prozedurale Typen

Eine Konstante prozeduralen Typs muß solche Prozeduren oder Funktionen referenzieren, die zuweisungskompatibel zum Typ der Konstanten sind. Ein Beispiel:

```
type
  ErrorProc = procedure(ErrorCode: Integer);

procedure DefaultError(ErrorCode: Integer); far;
begin
  WriteLn('Fehler ', ErrorCode, '.');
end;

const
  ErrorHandler: ErrorProc = DefaultError;
```

6. Ausdrücke

Ausdrücke bestehen aus Operatoren und Operanden. Die meisten Operatoren von Turbo Pascal verknüpfen zwei Operanden und werden deshalb als *binär* bezeichnet. Die restlichen Operatoren arbeiten mit nur einem Operanden, daher bezeichnet man sie als *unär*. Binäre Operatoren benutzen die herkömmliche algebraische Form wie z.B. *a* + *b*. Ein unärer Operator steht immer unmittelbar vor seinem Operanden, wie bei *-b*. Bei umfangreicheren Ausdrücken regelt die in Tabelle 6.1 gezeigte *Rangfolge* der Operatoren die Reihenfolge der Berechnung. Es gelten drei grundlegende Gesetze:

- Ein Operand zwischen zwei Operatoren von unterschiedlichem Rang ist immer an den höherrangigen Operator gebunden.
- Ein Operand zwischen zwei gleichrangigen Operatoren ist immer an den Operator gebunden, der links von ihm steht.
- Ausdrücke in Klammern werden als ein einzelner Operand betrachtet und immer als erstes ausgewertet.

Tabelle 6.1: Rangfolge der Operatoren

Operatoren	Rangfolge	Kategorie
@, not	1 (am höchsten)	unär
***, /, div, mod, and, shl, shr**	2	multiplizierend
+, -, or, xor	3	addierend
=, <>, <, >, <=, >=, in	4	relational

Operationen von gleichem Rang werden normalerweise von links nach rechts durchgeführt – der Compiler kann allerdings zur Optimierung des Rechenweges Umstellungen vornehmen.

Syntax von Ausdrücken

Die Rangfolge der Operatoren folgt der Syntax von Ausdrücken, die sich aus

Faktoren, Termen und einfachen Ausdrücken zusammensetzen. Faktoren lassen sich durch das auf der folgenden Seite gezeigte Syntaxdiagramm darstellen.

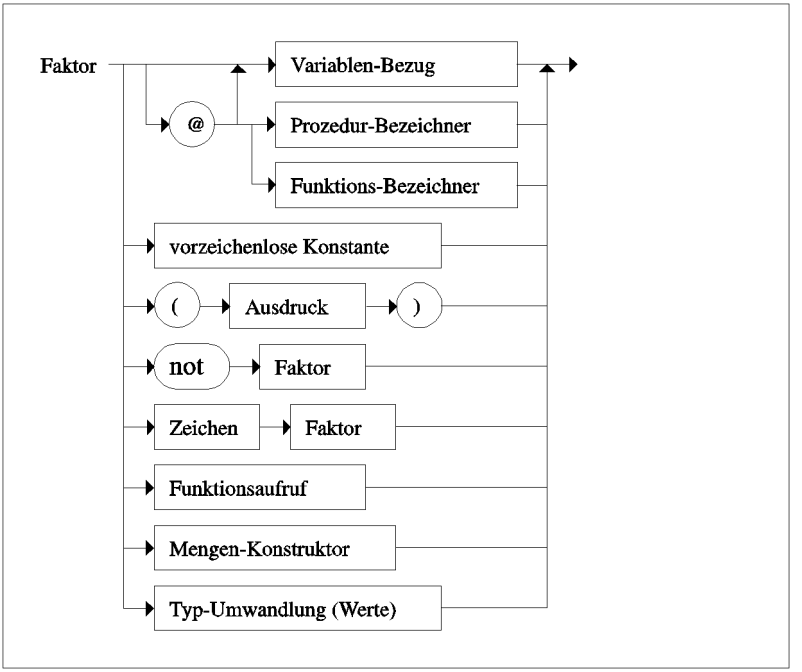


Abbildung 6.1 Syntax von Faktoren

Ein Funktionsaufruf aktiviert eine Funktion und kann deshalb mit dem von ihr zurückgelieferten Wert gleichgesetzt werden (vgl. Abschnitt »Funktionsaufrufe« in diesem Kapitel). Eine Mengenangabe steht für den Wert eines Mengen-Typs (vgl. »Mengen-Konstruktoren«). Eine Typumwandlung ändert den Typ eines Wertes. Eine vorzeichenlose Konstante hat folgende Syntax:

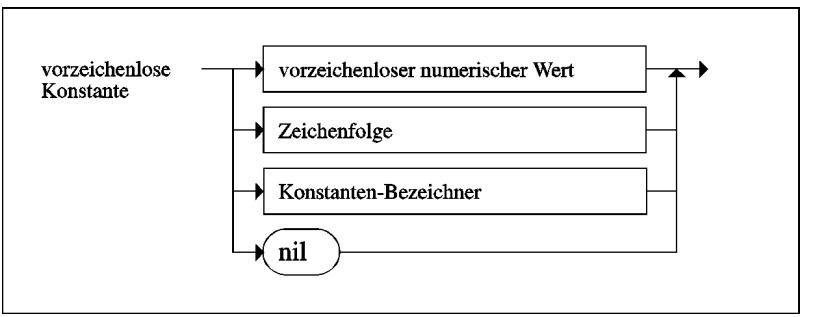


Abbildung 6.2 Syntax einer vorzeichenlosen Konstanten

Die folgenden Angaben stellen gültige Faktoren dar:

`X`	Variablenbezug
`MyUnit.X`	qualifizierter Variablenbezug
`@X`	Adresse einer Variablen
`@@MathPtr`	Adresse einer Prozedur-Variablen
`15`	vorzeichenlose Konstante
`(X+Y+Z)`	Unterausdruck
`Sin(X / 2)`	Funktionsaufruf
`MathPtr(X)`	Funktionsaufruf über Prozedur-Variable
`['0'..'9', 'A'.. 'Z']`	Mengenangabe
`not done`	Negation von Boolean
`Char(Digit+48)`	Umwandlung des Typs eines Wertes

Terme ordnen Faktoren Multiplikations-Operatoren zu:

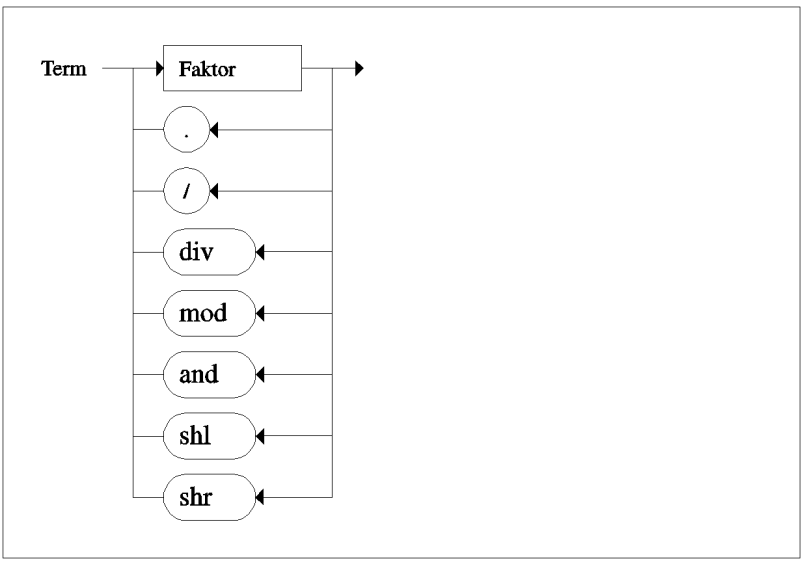


Abbildung 6.3 Syntax von Termen

Beispiele für gültige Terme sind:

```
X * Y
Z / (1-Z)
Done or Error
(X <= Y) and (Y < Z)
```

Einfache Ausdrücke ordnen Termen Additionsoperatoren und Zeichen zu:

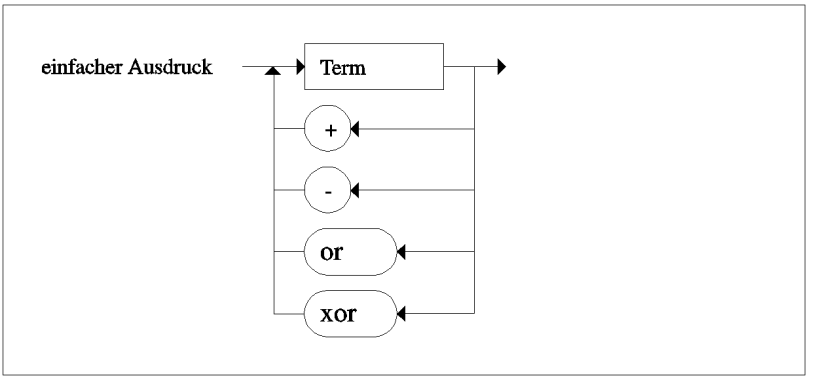


Abbildung 6.4 Syntax einfacher Ausdrücke

Einige Beispiele dazu:

```
X + Y
-X
Hue1 + Hue2
I*J + 1
```

Ein Ausdruck ordnet einfachen Ausdrücken relationale Operatoren zu:

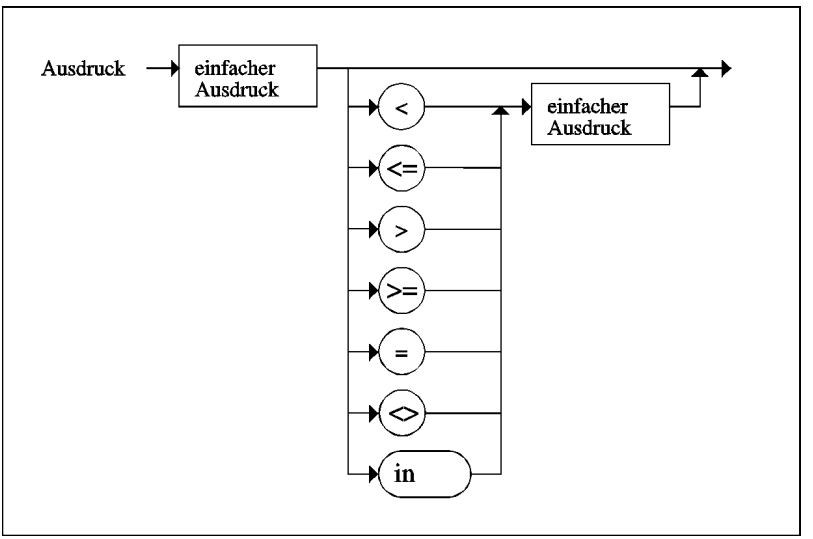


Abbildung 6.5 Syntax eines Ausdrucks

Einige Beispiele für Ausdrücke:

```
X = 1.5
Done <> Error
(I < J) = (J < K)
C in Hue1
```

Operatoren

Operatoren werden in sechs Gruppen unterschieden: arithmetische, logische, String-, Mengen-, relationale Operatoren und den Adreß-Operator **@**.

Arithmetische Operatoren

Die folgenden Tabellen zeigen die Operanden- und Ergebnistypen binärer bzw. unärer arithmetischer Operationen.

Tabelle 6.2 Binäre arithmetische Operationen

Operator	Operation	Operandentyp	Ergebnistyp
+	Addition	Integer	Integer
		Real	Real/Extended
-	Subtraktion	Integer	Integer
		Real	Real/Extended
*	Multiplikation	Integer	Integer
		Real	Real/Extended
/	Division	Integer	Integer
		Real	Real/Extended
div	Integerdivision	Integer	Integer
mod	Modulo	Integer	Integer

Hinweis: Der Operator + wird auch als Stringoperator benutzt, die Operatoren +, - und * werden auch als Mengenoperatoren benutzt.

Tabelle 6.3: Unäre arithmetische Operationen

Operator	Operation	Operandentyp	Ergebnistyp
+	Identität	Integer	Integer
		Real	Real/Extended
-	Negation	Integer	Integer
		Real	Real/Extended

Operanden, die Teilbereiche ordinaler Typen darstellen, werden auf dieselbe Weise wie der »ursprüngliche« Ordinaltyp behandelt. Wenn beide Operanden eines Operators +, -, *, div oder mod einen Integer-Typ haben, hat das Ergebnis den Typ des gemeinsamen Formats (siehe Abschnitt »Der Typ Integer« in Kapitel 3 »Typen«).

Ist einer der Operanden eines Operators +, - oder * vom Typ *Real*, dann hat das Ergebnis im Modus `$N-` den Typ *Real*, im Modus `$N+` den Typ *Extended*. Ist der Operand eines Identitäts- oder Negations-Operators ein Integer-Typ, dann ist auch das Ergebnis im selben Format. Ist der Operator dagegen ein Real-Typ, dann ergibt sich – je nach Stand des Schalters `$N` – ein Wert vom Typ *Real* oder *Extended*. Das Ergebnis von *x/y* ist immer vom Typ *Real* bzw. *Extended*, wobei der Typ der Operanden keine Rolle spielt. Wenn *y* den Wert 0 hat, ergibt sich ein Laufzeitfehler.

Der Wert von *i* **div** *j* ist der mathematische Quotient von *i/j*, auf einen Integerwert abgerundet. Wenn *j* den Wert 0 hat, ergibt sich ein Laufzeitfehler.

Der Operator **mod** liefert den Rest der Division seiner beiden Operanden zurück, also:

```
i mod j = i - (i div j) * j
```

Das Ergebnis einer Operation mit **mod** übernimmt das Vorzeichen seines Operanden *i*. Wenn *j* den Wert 0 hat, ergibt sich ein Laufzeitfehler.

Logische Operatoren

Tabelle 6.4 gibt die Typen der Operanden und Ergebnisse logischer Operationen wieder.

Tabelle 6.4: Logische Operationen

Operator	Operation	Operandentyp	Ergebnistyp
not	**bitweise Negation**	**Integer**	**Integer**
and	**bitweises UND**	**Integer**	**Integer**
or	**bitweises ODER**	**Integer**	**Integer**
xor	**bitweise Antivalenz**	**Integer**	**Integer**
shl	**Linksschieben**	**Integer**	**Integer**
shr	**Rechtsschieben**	**Integer**	**Integer**

Hinweis: **not** ist ein unärer Operator.

Wenn der Operand von **not** einen Integer-Typ hat, dann ist auch das Ergebnis in diesem Format.

Wenn beide Operanden von **and**, **or** oder **xor** einen Integer-Typ haben, dann hat das Ergebnis das gemeinsame Format der beiden Operanden (vgl. Kapitel 3 »Typen«).

Die Operationen *i* **shl** *j* und *i* **shr** *j* verschieben den Wert von *i* um *j*

Bitpositionen nach links bzw. nach rechts, entsprechen also einer Multiplikation bzw. Division mit 2^i. Der Typ des Ergebnisses ist der Typ von i.

Boolesche Operatoren

Tabelle 6.5 gibt die Typen der Operanden und Ergebnisse boolescher Operationen wieder.

Tabelle 6.5: Boolesche Operationen

Operator	Operation	Operandentyp	Ergebnistyp
not	logische Negation	Boolean	Boolean
and	logisches UND	Boolean	Boolean
or	logisches ODER	Boolean	Boolean
xor	logische Antivalenz	Boolean	Boolean

Hinweis: Der Operator **not** ist auch hier unär.

Bei Operanden des Typs *Boolean* bestimmt die normale boolesche Logik das Ergebnis dieser Operationen. Beispielsweise ergibt *a* **and** *b* nur dann *True* (wahr), wenn *a* und *b* wahr sind.

Turbo Pascal unterstützt zwei Arten der Code-Erzeugung für die Auswertung zusammengesetzter Ausdrücke mit den Operatoren **and** und **or**: vollständige Berechnung und das *Kurzschlußverfahren*:

- Bei einer vollständigen Berechnung werden sämtliche Teile des Ausdrucks berechnet – selbst dann, wenn das Ergebnis bereits feststeht, also beispielsweise der erste Teil einer **and**-Operation den Wert *False* ergibt. Diese Art der Auswertung ist z.B. erforderlich, wenn Teile des Ausdrucks aus Funktionsaufrufen bestehen, über die weitere Operationen ausgeführt und/oder Werte verändert werden (Seiteneffekt).

- Das Kurzschlußverfahren garantiert eine strikte Auswertung von links nach rechts und bricht ab, sobald das Ergebnis des Gesamtausdrucks feststeht – beispielsweise dann, wenn der erste Teil einer **or**-Operation den Wert *True* ergibt. Die Art der Auswertung erhöht nicht nur die Arbeitsgeschwindigkeit des Programms und erzeugt normalerweise kompakteren Code, sondern ermöglicht auch Konstrukte, die bei vollständiger Berechnung nicht legal wären.

Zwei Beispiele dazu:

```
while (I <= Length(S)) and (S[I] <> ' ') do Inc(I);
while (P <> nil) and (P^.Wert <> 5) do P := P^.Next;
```

Hier wird in beiden Fällen beim Kurzschlußverfahren die zweite Prüfung nicht durchgeführt, wenn die erste den Wert *False* ergibt. Bei einer vollständigen Berechnung ergäben sich Zugriffe auf nicht definierte Adressen als Teil der

Prüfung selbst – die Bedingungen müßten deshalb folgendermaßen formuliert werden:

```
while (I <= Length(S)) do
    if S[I] <> ' ') then Inc(I);
while (P <> nil) do
    if (P^.Wert <> 5) then P := P^.Next;
```

Welche Art von Code der Compiler zur Auswertung zusammengesetzter Bedingungen generiert, wird über den Schalter `$B` kontrolliert, der standardmäßig auf `$B-` (Kurzschlußverfahren) gesetzt ist. Nur wenn dieser Schalter (mit `$B+` oder via *Options/Compiler*) aktiviert ist, werden zusammengesetzte logische Ausdrücke vollständig berechnet.

Da Standard-Pascal nicht definiert, in welcher Weise logische Ausdrücke ausgewertet werden, sind Programme, die sich auf eines der beiden Verfahren verlassen, nicht vollständig portabel. Die Formulierung zusätzlicher Bedingungen (wie oben gezeigt), stellt den sichereren Weg dar, der aber aufwendiger und bei weitem nicht so übersichtlich ist.

String-Operatoren

Tabelle 6.6 gibt die Typen der Operanden und Ergebnisse von String-Operationen wieder.

Tabelle 6.6: String-Operationen

Operator	Operation	Operandentyp	Ergebnistyp
+	Verbindung	String, Char oder packed String	String

In Turbo Pascal kann der Operator + verwendet werden, um zwei String-Operanden miteinander zu verbinden. Das Ergebnis einer Operation wie *s* + *t* ist die Verbindung von *s* und *t*, wobei *s* und *t* entweder ein String-Typ, ein Char-Typ oder ein gepackter String-Typ sein müssen. Das Resultat ist mit jedem String-Typ kompatibel (allerdings nicht mit den Typen *Char* und **packed** *String*). Wenn das Ergebnis die Länge von 255 Zeichen überschreitet, wird es nach dem 255. Zeichen abgeschnitten.

PChar-Operatoren

Ist die erweiterte Syntax über den Compiler-Befehl **{$X+}** eingeschaltet, wird eine Reihe neuer Operatoren für Werte des Typs *PChar* unterstützt. Die Operatoren + und - können dazu verwendet werden, den *Offset*-Wert eines solchen Zeigers zu verändern. Über den Minus-Operator kann der Unterschied der

Offset-Werte zweier solcher Zeiger ermittelt werden. Angenommen, *P* und *Q* sind Werte des Typs *PChar* und *I* ein Wert des Typs Word, dann sind folgende Operationen möglich:

P + I	I zum Offset von P addieren;
I + P	I zum Offset von P addieren;
P - I	I vom Offset von P subtrahieren;
P - Q	Offset von Q vom Offset von P subtrahieren

Die Operationen *P* + *I* und *I* + *P* addieren *I* zum Offset-Anteil von *P*, der darus resultierende Zeiger zeigt auf *I* nach *P*. Die Operation *P* - *I* subtrahiert *I* vom Offset-Anteil von *P*, der darus resultierende Zeiger zeigt auf *I* vor *P*.

Die Operation *P* - *Q* berechnet die Distanz zwischen *Q* (der niedrigeren Adresse) und *P* (der höheren Adresse). Das Resultat, ein Wert des Typs Word, entspricht der Anzahl der Zeichen zwischen *Q* und *P*. Diese Operation geht davon aus, daß *P* und *Q* in das gleiche Array des Typs **array**[0..X] **of** Char zeigen. Ist das nicht der Fall, ist das Ergebnis der Operation undefiniert.

Mengen-Operatoren

Tabelle 6.7 gibt die Operatoren-Typen für Operationen mit Mengen wieder.

Tabelle 6.7: Mengen-Operationen

Operator	Operation	Operandentyp
+	**Vereinigung**	**kompatible Mengen**
-	**Differenz**	**kompatible Mengen**
*****	**Durchschnitt**	**kompatible Mengen**

Die Resultate der Operationen ergeben sich durch die Gesetze der Mengenlehre:

- ein Ordinalwert *c* ist nur dann in *A*+*B* enthalten , wenn er in *A* oder *B* enthalten ist.
- ein Ordinalwert *c* ist nur dann in *A*-*B* enthalten, wenn er in *A*, aber nicht in *B* enthalten ist.
- ein Ordinalwert *c* ist nur dann in *A***B* enthalten, wenn er sowohl in *A* als auch in *B* enthalten ist.

Wenn *a* als Element des Ergebnisses einer Mengen-Operation der kleinste Ordinalwert innerhalb dieser Menge ist und *b* der größte, dann ist **set of** *a..b* der Typ des Ergebnisses.

Relationale Operatoren

Tabelle 6.8 gibt die Operandentypen und die Ergebnisse relationaler Operationen wieder.

Tabelle 6.8: Relationale Operationen

Operator	Operation	Operandentyp	Ergebnistyp
=	gleich	kompatible einfache, Zeiger-, Mengen- String- oder gepackte String-Typen	Boolean
<>	ungleich	kompatible einfache, Zeiger-, Mengen- String- oder gepackte String-Typen	Boolean
<	kleiner als	kompatible einfache, Zeiger-, Mengen- String- oder gepackte String-Typen>	Boolean
	größer als	kompatible einfache, Zeiger-, Mengen- String- oder gepackte String-Typen	Boolean
<=	kleiner/gleich	kompatible einfache, Zeiger-, Mengen- String- oder gepackte String-Typen	Boolean
>=	größer/gleich	kompatible einfache, Zeiger-, Mengen- String- oder gepackte String-Typen	Boolean
<=	Untermenge von	kompatible Mengen-Typen	Boolean
>=	Obermenge von	kompatible Mengen-Typen	Boolean
in	Element von	linker Operand: beliebiger Ordinaltyp t rechter Operand: set of Typ t	Boolean

Vergleiche einfacher Typen

Wenn die Operanden der Operatoren =, <>, <, >, <= oder >= einfache Typen haben, müssen diese Typen jeweils kompatibel sein. Wenn allerdings einer der Operanden vom Real-Typ ist, kann der andere den Typ *Integer* haben.

Vergleiche von Strings

Die relationalen Operatoren =, <>, <, >, <= und >= vergleichen Strings nach der durch die Anordnung des IBM-Zeichensatzes vorgegebenen Reihenfolge. Es können alle Stringwerte miteinander verglichen werden, weil bekanntlich alle Stringwerte zueinander kompatibel sind.

Ein Char-Typ ist zu einem String-Typ kompatibel und wird bei Vergleichen als String mit der Länge 1 bewertet. Beim Vergleich eines gepackten String-Typs mit *n* Komponenten eines anderen String-Typs wird der gepackte String als Wert eines String-Typs mit der Länge *n* bewertet.

Vergleiche von gepackten Strings

Die relationalen Operatoren =, <>, <, >, <= und >= können ebenfalls dazu verwendet werden, um zwei Werte von gepackten String-Typen miteinander zu vergleichen, wenn beide die gleiche Anzahl an Komponenten haben. In diesem Fall entspricht die Operation dem Vergleich zweier ungepackter Strings mit derselben Anzahl von Komponenten.

Vergleiche von Zeigern

Die Operatoren = und <> können benutzt werden, um kompatible Zeiger miteinander zu vergleichen. Zwei Zeiger sind nur dann gleich, wenn sie auf dasselbe Objekt zeigen.

Hinweis: Turbo Pascal vergleicht bei Zeigern lediglich die Einzelwerte von *Segment* und *Offset* miteinander. Aufgrund der Mehrdeutigkeit von Adressen bei Prozessoren der Reihe 80x86 können deshalb zwei logisch »verschiedene« Zeiger auf ein und dieselbe physikalische Adresse zeigen – Wertepaare wie $0040:$0049 und $0000:$0449 sind technisch gesehen identisch. Die Prozeduren *New* und *GetMem* liefern Zeiger immer in normalisierter Form zurück, d.h. mit einem *Offset*-Anteil im Bereich von $0000 bis $000F – hier haben Vergleiche immer das »richtige« Ergebnis. In allen anderen Fällen (z.B. bei der Wertzuordnung über die Funktion *Ptr*) muß explizit normalisiert bzw. auf möglicherweise falsche Ergebnisse von Vergleichen geachtet werden.

Vergleiche von PChar-Zeigern

Ist die erweiterte Syntax (über den Compiler-Befehl **{$X+}**) eingeschaltet, sind die Operatoren >, <, >= und <= auch auf Werte des Typs *PChar* anwendbar. Beachten Sie aber, daß solche Vergleichsoperationen davon ausgehen, daß die miteinander verglichenen Zeiger in den *gleiche* Zeichenarray zeigen. Daher vergleichen diese Operatoren lediglich die *Offset*-Werte der Zeiger. Zeigen diese aber auf verschiedene Zeichenarrays, ist das Ergebnis des Vergleichs undefiniert.

Vergleiche von Mengen

Wenn *a* und *b* Mengen sind, erzeugt ihr Vergleich folgende Resultate:

- *A* = *B* ist nur dann wahr, wenn *A* und *B* exakt dieselben Elemente enthalten. In jedem anderen Fall ist *A* <> *B*.
- *A* <= *B* ist nur dann wahr, wenn alle Elemente von *A* auch Elemente von *B* sind.
- *A* >= *B* ist nur dann wahr, wenn alle Elemente von *B* auch Elemente von *A* sind.

Überprüfung der Mengenzugehörigkeit

Der Operator **in** liefert den Wert *True* zurück, wenn der Wert des linken Operanden ein Ordinal-Typ und ein Element des rechten Operanden (d.h. der Menge) ist; andernfalls wird der Wert *False* zurückgeliefert.

Der Adreß-Operator @

Über den Adreß-Operator @ kann die Adresse einer Variablen bestimmt und einem Zeiger zugeordnet werden. Tabelle 6.9 zeigt mögliche Operanden und Ergebnistypen.

Tabelle 6.9: Zeiger-Operationen

Operator	Operation	Operandentyp	Ergebnistyp
@	Adreß-Ermittlung	Variablenbezug, Funktions- oder Prozedur-Bezeichner	Pointer (derselbe Variablentyp wie **nil**)

@ ist ein unärer Operator. Als Operanden verwendet er einen Variablen-Bezug bzw. einen Prozedur- oder Funktionsbezeichner und liefert einen Zeiger auf diesen Operanden zurück (d.h. eine Adresse). Der Typ dieses Wertes entspricht dem des Typs von **nil** (»untypisierter Zeigerwert«) und kann deshalb jeder beliebigen Zeiger-Variablen zugewiesen werden.

Hinweis: Da eine prozedurale Variable zum einen eine Adresse enthält, zum anderen aber selbst eine Adresse hat, wurde die Syntax von @ entsprechend erweitert: Wenn *F* eine prozedurale Variable ist, dann liefert **@***F* die Adresse der Routine, die *F* zugeordnet ist; **@@F** liefert die Adresse, ab der die Variable *F* selbst im Speicher steht (vgl. »Prozedurtypen« in Kapitel 3 »Typen« und Kapitel 8 »Prozeduren und Funktionen«).

@ mit einer Variablen

Gegeben seien folgende Deklarationen:

```
type
  ZweiChar = packed array[0..1] of Char;
var
  Int : Integer;
  ZweiCharPtr : ^ZweiChar;
```

Die folgende Anweisung weist nun dem Zeiger *ZweiCharPtr* die Adresse der Variablen *Int* zu:

```
ZweiCharPtr := @Int;
```

Da *ZweiCharPtr* als Zeiger auf ein Char-Array definiert wurde, läßt sich der Wert von *Int* nun ebenfalls als **packed array***[0..1]* of *Char* interpretieren: *ZweiCharPtr^[0]* ist ein Bezug auf das niederwertige, *ZweiCharPtr^[1]* ein Bezug auf das höherwertige Byte von *Int*.

@ mit einem Wert-Parameter

Wenn @ auf einen formalen Wert-Parameter angewendet wird, ergibt sich ein Zeiger auf die Adresse innerhalb des Stacks, die den aktuellen Wert enthält, also

```
procedure WertZeiger (Parm1: Integer);
var
  PPtr: ^Integer;
begin
  PPtr := @Parm1;
  .
```

```
  .
end;
```

Diese Adreßermittlung ist nicht mit einem Variablen-Parameter vergleichbar – da die Routine nur eine Kopie des jeweils aktuellen Parameterwertes übergeben bekommt, bezieht sich *PPtr^* nach der gezeigten Zuweisung nicht auf eine »übergebene Variable«, sondern auf eine Kopie ihres Wertes, die sich auf dem Stack befindet. Eine Zuordnung wie *PPtr^ := 12;* verändert also lediglich den übergebenen Wert – nicht etwa eine Variable, mit der die Prozedur aufgerufen wurde.

@ mit einem Variablen-Parameter

Wenn @ auf einen formalen Variablen-Parameter angewendet wird, ergibt sich ein Zeiger auf den aktuellen Parameter, dessen Adresse vom Stack gelesen wird, also:

```
procedure VarZeiger (var Parm1: Integer);
var
  PPtr: ^Integer;
begin
  PPtr := @Parm1;
  .
  .
end;
```

In diesem Fall bezieht sich *PPtr^* tatsächlich direkt auf die übergebene Variable, die durch eine Zuordnung wie *PPtr^ := 12;* entsprechend verändert wird.

@ mit Prozeduren und Funktionen

Die Anwendung des Operators @ auf den Bezeichner einer Prozedur oder Funktion liefert die Startadresse der entsprechenden Routine zurück. Turbo Pascal enthält keine weiteren Mechanismen zur Verwendung dieses Wertes, der sich – im Gegensatz zu Prozedur-Variablen – jeder Kontrolle des Compilers entzieht. Der einzige Zweck der Anwendung von @ auf Routinen besteht in der Übergabe von Adressen an Maschinensprache-Unterprogramme (siehe Kapitel 22 »Der integrierte Assembler« und Kapitel 23 »Einbinden von Assembler-Code«).

@ mit Methoden

Der Operator @ kann auf qualifizierte Methoden-Bezeichner angewandt werden, wenn der Einsprungspunkt der Methode gebraucht wird.

Funktions-aufrufe

Ein Funktionsaufruf aktiviert die Funktion, die durch den angegebenen Bezeichner spezifiziert ist. (Jeder Bezeichner, der zur Repräsentation einer Funktion deklariert wird, ist ein Funktions-Bezeichner).

Wenn die dazugehörige Funktions-Deklaration eine Liste formaler Parameter enthält, muß der Funktionsaufruf analog dazu eine Liste aktueller Parameter angeben. Jeder aktuelle Parameter nimmt dabei jeweils den Platz eines formalen Parameters ein (siehe Kapitel 8 »Prozeduren und Funktionen«).

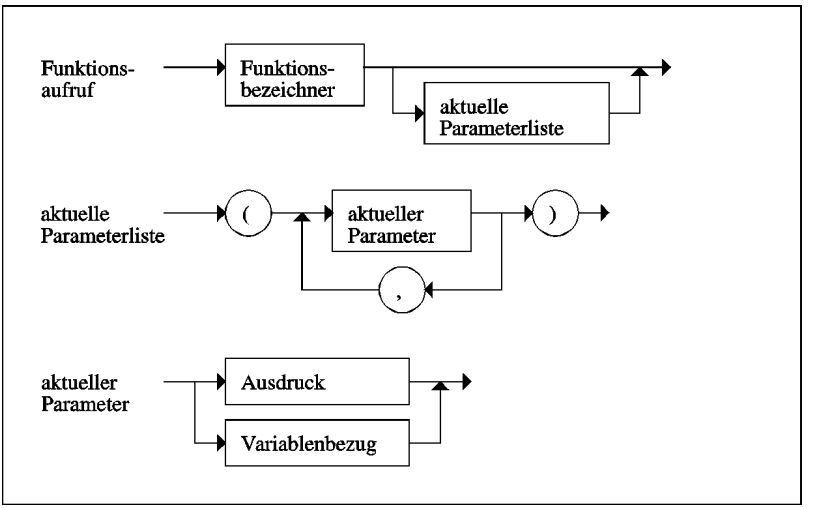


Abbildung 6.6 Syntax von Funktionsaufrufen

Einige exemplarische Funktionsaufrufe:

```
Summe(A,639)
Maximum(143, J)
Sin(x + y)
Eof(F)
```

Funktionen können auch über prozedurale Variablen aktiviert werden (vgl. Abschnitt »Prozedurale Typen« in Kapitel 8 »Prozeduren und Funktionen«).

Weiterhin erlaubt die Syntax von Funktionsaufrufen, daß der Aufruf eines Methodenbezeichners anstelle eines Funktionsaufrufs verwendet werden kann.

Die Besprechung der Erweiterungen für Prozeduranweisungen in Kapitel 7 »Anweisungen« gilt auch für Funktionsaufrufe.

Im {$X+}-Modus (Erweiterte Syntax) können Funktionsaufrufe wie Anweisungen gebraucht werden; das heißt, deren Funktionswert bleibt unausgewertet. Detaillierte Informationen hierzu finden Sie in Kapitel 21 »Compiler-Befehle«.

Mengen-Konstruktoren

Eine Mengenangabe repräsentiert den Wert des jeweiligen Mengen-Typs und besteht aus Ausdrücken, die von eckigen Klammern umschlossen sind. Jeder dieser Ausdrücke repräsentiert ein Element der Menge.

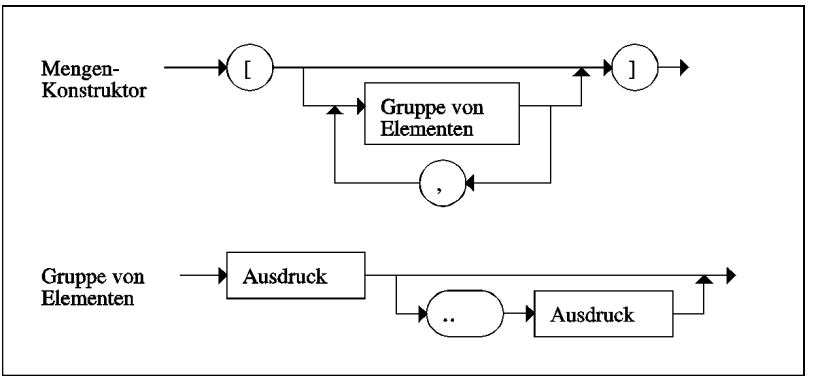


Abbildung 6.7 Syntax von Mengen-Konstruktoren

Der Ausdruck *[]* steht für die leere Menge, die zu jedem Mengen-Typ zuweisungskompatibel ist. Eine Bereichsangabe wie *x..y* repräsentiert sämtliche Elemente einer Menge, deren ordinale Werte größer/gleich dem ordinalen Wert von *x* und kleiner/gleich dem ordinalen Wert von *y* sind. Wenn *x* größer ist als *y*, dann steht *x..y* für *kein* Element der Menge, die Angabe *[x..y]* ist in diesem Fall mit *[]* identisch.

Jeder Wert jedes Ausdrucks einer Gruppe von Elementen in einer Mengenangabe muß denselben Ordinal-Typ haben. Beispiele für Mengenangaben sind:

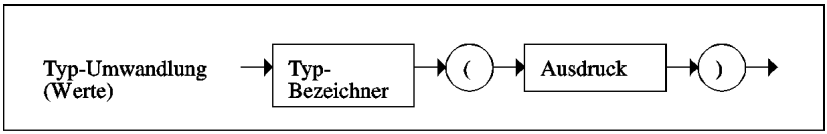
```
[rot, C, gruen]
['A'..'Z', 'a'..'z', Chr(Digit+48)]
[1, 5, 10..K mod 12, 13]
```

Umwandlung von Wert-Typen

Über eine Wert-Umwandlung (value-type cast) kann der Typ eines Ausdrucks in einen anderen Typ umgewandelt werden.

Das Argument des Ausdrucks muß entweder ein Ordinal- oder ein Zeiger-Typ sein. Das Ergebnis hat den spezifizierten Typ, sein Ordinalwert ergibt sich aus der Umwandlung des Ausdrucks. Diese Umwandlung kann ein Abschneiden oder auch eine Erweiterung des ursprünglichen Wertes mit sich bringen, wenn

die Größe des spezifizierten Typs ungleich der des Ausdrucks ist. Bei Erweiterungen von Werten bleibt das Vorzeichen erhalten.



Die Syntax zur Umwandlung eines Wert-Typs ist zu der Umwandlung eines Variablen-Typs nahezu äquivalent (siehe Kapitel 4 »Variablen«). Allerdings arbeitet diese Umwandlung mit Werten, nicht mit Variablen – deshalb kann sie nicht bei der Bezugnahme auf Variablen benutzt werden (woraus folgt, daß auf die Typ-Umwandlung eines Wertes keine Qualifizierer folgen können). Diese Art der Umwandlung darf nicht auf der linken Seite einer Zuweisung erscheinen. Gültige Beispiele für Umwandlungen von Wert-Typen sind:

```
Integer('A')
Char(48)
Boolean(0)
Farbe(2)
Longint(@Buffer)
BytePtr(Ptr($40,$49))
```

Prozedur-Variablen in Ausdrücken

Wenn eine Prozedur-Variable in einem Ausdruck oder einer Anweisung erscheint, wird sie normalerweise mit einem Aufruf der entsprechenden Routine gleichgesetzt. Von dieser Regel muß es eine Ausnahme geben: Wenn der Compiler auf der *linken* Seite einer Zuweisung eine Prozedur-Variable erkennt, behandelt er die rechte Seite nicht als Aufruf, sondern als Wert – anders wäre die Zuweisung einer Routine an eine Variable oder die Zuweisung zwischen Prozedur-Variablen auch nicht möglich.

Ein Beispiel dazu:

```
type
  IntFunc = function: Integer;

var
  F: IntFunc;
  N: Integer;

function ReadInt: Integer; far
var
  I: Integer;
begin
```

```
  Readln(I);
  ReadInt := I;
end;

begin
  F := ReadInt;   { Zuweisung an die Prozedur-Variable }
  N := ReadInt;   { Aufruf der Funktion ReadInt }
  N := F;         { dito }
end.
```

Die erste Zuweisung des Hauptprogramms enthält auf der linken Seite eine Prozedur-Variable – folglich codiert der Compiler hier keinen Aufruf der Funktion *ReadInt*, sondern weist diese Funktion der Variablen *F* zu (technisch: läßt die Adresse von *ReadInt* in *F* speichern).

Während sich die Frage »Aufruf oder Funktionsadresse« bei Zuweisungen immer eindeutig beantworten läßt, stellen Vergleiche den Compiler vor ein Kompatibilitätsproblem. In einer Anweisung wie

```
if F = ReadInt then Writeln('Ist gleich.');
```

ließe sich zwar ebenfalls (über den »Typ der linken Seite«) festlegen, daß hier die momentan in *F* gespeicherte Adresse mit der von *ReadInt* verglichen werden soll – das aber läuft der Sprachdefinition von Pascal zuwider, in der festgelegt ist, daß ein Funktionsbezeichner in einem *Ausdruck* gleichbedeutend mit dem Aufruf dieser Funktion ist. Turbo Pascal hält sich an diesen Standard: Der Compiler würde also im gezeigten Beispiel einen Aufruf von *ReadInt* codieren und dann versuchen, das Ergebnis dieser Funktion mit der Variablen *F* zu vergleichen (was mit einer Beschwerde über unpassende Typen endet).

Für den *Vergleich* einer Prozedur-Variablen mit der Adresse einer Routine wird deshalb eine Sonderform benutzt:

```
if @F = @ReadInt then Writeln('F zeigt auf die Funktion ReadInt');
```

Der Adreß-Operator **@** steht hier für den *Inhalt* der Prozedur-Variablen *F* und macht die Absicht des Programmierers eindeutig. Technisch gesehen, steht er für eine Umwandlung in eine untypisierte Variable des Typs *Pointer* – und die läßt sich mit der (ebenfalls untypisierten) Adresse einer Funktion ohne Kompatibilitätsprobleme vergleichen.

Als Konsequenz dieser Festlegung wird die *Adresse* einer Prozedur-Variablen über die doppelte Anwendung des Operators @ ermittelt: Wenn *F* als Prozedur-Variable deklariert ist, dann liefert **@F** den in dieser Variablen gespeicherten Wert (d.h. eine Routinen-Adresse in Form eines untypisierten Zeigers); **@@F** liefert die Adresse, ab der die Variable *F* selbst im Speicher steht.

7. Anweisungen

Der englische Begriff *Statement* steht für alle Konstrukte, die eine durch den Computer ausführbare Aktion vereinbaren. Wir verwenden in diesem Handbuch den Terminus »Anweisung« als Oberbegriff für Anweisungen (wie **begin**, **end** oder **for**) und *Befehle* (wie **goto**, Zuweisungen, Prozeduraufrufe usw.).

Jeder Anweisung (d.h. jeder Vereinbarung einer ausführbaren Aktion) kann ein *Label* vorangestellt werden, auf das wiederum ein Bezug mit **goto** möglich ist: ein **goto** zu diesem Label bewirkt einen direkten Sprung zu dieser Anweisung und ihre Ausführung.

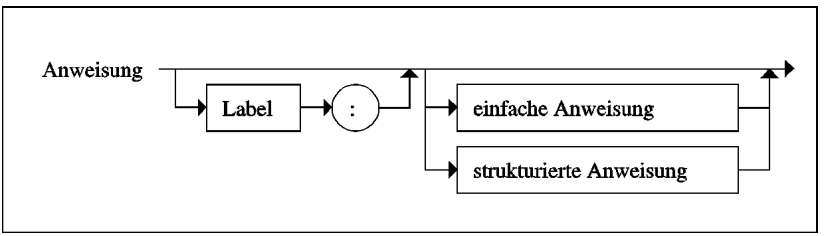


Abbildung 7.1 Syntax der goto-Anweisung

Wie bereits in Kapitel 1 »Sprachelemente und Konstanten« definiert, ist ein Label entweder eine Ziffernfolge im Bereich von 0 bis 9999 oder ein Bezeichner. Anweisungen werden nach den Kriterien *einfach* und *strukturiert* unterschieden:

Einfache Anweisungen

bestehen aus Zuweisungen, Prozeduraufrufen oder Sprüngen, d.h. enthalten in sich keine weiteren Anweisungsebenen.

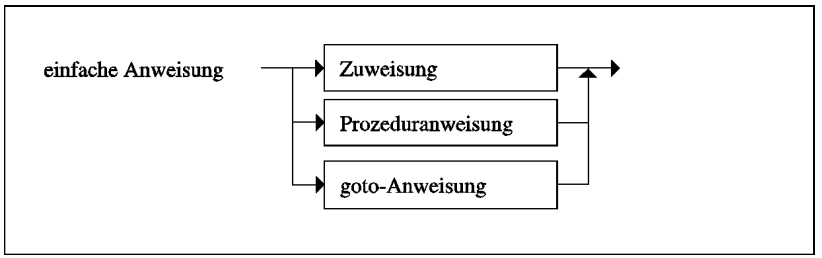


Abbildung 7.2 Syntax einer einfachen Anweisung

Zuweisungen

Zuweisungen ersetzen den momentanen Wert einer Variablen mit einem neuen Wert, der über einen Ausdruck angegeben wird. Dieser Ausdruck kann Bezeichner von Funktionen enthalten, die dadurch aktiviert werden und entsprechende Werte zurückliefern.

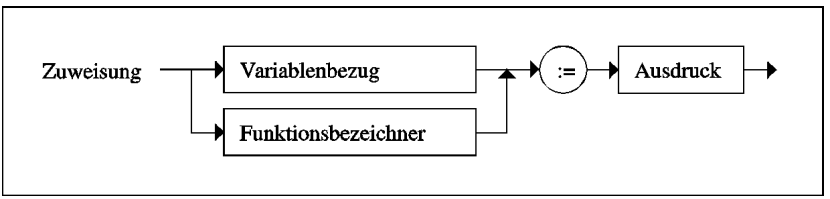


Abbildung 7.3 Syntax einer Zuweisung

Der Ausdruck muß zum Typ der Variablen oder zum Ergebnistyp der Funktion zuweisungskompatibel sein (vgl. Abschnitt »Kompatibilität von Typen« in Kapitel 4 »Variablen«). Gültige Zuweisungen sind:

```
X := Y + Z;
Done := (I>=1) and (I<100);
Farben1 := [blau, Succ(Farbe)];
I := Sqr(J) - I*K;
```

Die Regeln für die Zuweisungskompatibilität ermöglichen, daß der Instanz eines Objekttyps die Instanz eines ihrer Nachkommenstypen zugewiesen wird. Eine derartige Zuweisung stellt eine *Projektion* des Nachkommens auf den Raum dar, der von seinen Vorfahren umspannt wird. Wenn zum Beispiel eine Instanz *F* des Typs *Field* und eine Instanz *Z* des Typs *ZipField* vorliegt, dann werden mit der Zuweisung *F := Z* nur die Felder *X, Y, Len* und *Name* kopiert.

Die Zuweisung einer Instanz eines Objekttyps ist *nicht* mit einer Initialisierung der Instanz verbunden. Auf das vorherige Beispiel bezogen, bedeutet die Zuweisung *F := Z* nicht, daß der Konstruktoraufruf für *F* weggelassen werden kann.

Prozedur-aufrufe

Durch Angabe eines Prozedurbezeichners wird eine Prozedur aufgerufen, die mit diesem Bezeichner deklariert wurde. Wenn die Deklaration der Prozedur eine Liste von formalen Parametern enthält, dann muß beim Aufruf eine korrespondierende Liste aktueller Parameter angegeben werden. (Alle Parameter, die im Deklarationsteil einer Prozedur aufgelistet sind, bezeichnet man als *formale* Parameter – die entsprechenden Parameter in den Aufrufen innerhalb des Anweisungsteils eines Programms werden dagegen *aktuelle* Parameter genannt). Die Übergabe der aktuellen Parameter an die formalen Parameter ist Bestandteil des Aufrufs.

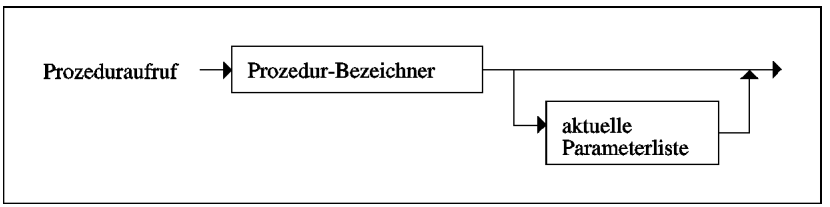


Abbildung 7.4 Syntax eines Prozeduraufrufs

Auch hierzu einige Beispiele (wobei vorausgesetzt wird, daß die entsprechenden Prozeduren an anderer Stelle des Programms deklariert sind):

```
PrintHeading;
Transpose (A,M,J);
Find(Name, Adresse);
```

Hinweis: Als Erweiterung des Sprachstandards erlaubt Turbo Pascal die Aktivierung von Prozeduren nicht nur durch die Angabe ihres Bezeichners, sondern auch über prozedurale Variablen (siehe Kapitel 8 »Prozeduren und Funktionen«).

Die Syntax einer Prozeduranweisung erlaubt auch, daß ein Methodenbezeichner für eine Prozedur, einen Konstruktor oder einen Destruktor anstelle eines Prozedur-Bezeichners verwendet werden kann.Methode:Aufruf als Funktion oder Prozeodur;

Die Instanz eines Methodenbezeichners dient zweierlei Zwecken. Erstens, steht im Falle einer virtuellen Methode der *tatsächlich* (zur Laufzeit) vorliegende Typ der Instanz fest, so daß klar ist, welche Implementierung aktiviert wird. Zweitens wird die Instanz selbst implizit zu einem aktuellen Parameter der Methode. Sie entspricht einem formalen Variableneintrag mit der Bezeichnung *Self*, die den Typ der aktivierten Methode aufweist.

Innerhalb einer Methode kann man in einer Prozeduranweisung einen vollständigen Methodenbezeichner angeben, um die Aktivierung einer bestimmten Methode zu erzwingen. Der Objekttyp im vollständigen Bezeichner muß derselbe wie der Objekttyp der Methode oder einer seiner Vorfahren sein. Diese Art von Aktivierung wird als *vollständige Aktivierung* bezeichnet.

Der implizite Parameter *Self* einer vollständigen Aktivierung wird zum Parameter *Self* der Methode, die den Aufruf enthält. Eine vollständige Aktivierung verwendet niemals den Aufrufmechanismus der virtuellen Methoden. Der Aufruf erfolgt immer statisch, es wird immer die angegebene Methode aufgerufen.

Eine Aktivierung mit vollständigen Bezeichnern wird im allgemeinen in einer Ersatzmethode verwendet, um die ersetzte Methode aufrufen zu können. Bezogen auf die bereits früher deklarierten Typen folgen ein paar Beispiele mit vollständigen Bezeichnern:

```
constructor NumField.Init(FX, FY, FLen : Integer;
  FName : String; FMin, FMax : Longint);
begin
  Field.Init(FX, FY, FLen, FName);
  Wert := 0;
  Min := FMin;
  Max := FMax;
end;

function ZipField.PutStr(S : String) : Boolean;
begin
  PutStr := (Length(S) = 5) and NumField.PutStr(S);
end;
```

Wie diese Beispiele zeigen, kann man mit einem vollständigen Aufruf in einer Ersatzmethode den Programmcode der Methode, die man ersetzt, »wiederverwenden«. Die Methode des Vorfahren kann also auch in der Methode des Nachkommen aufgerufen werden, auch wenn letztere die Methode des Vorfahren ersetzt.

Die Anweisung goto

führt einen Sprung zum angegebenen Label aus: Das Programm wird an dem Punkt fortgesetzt, der unmittelbar auf das Label folgt. Die Syntax von **goto** ist:

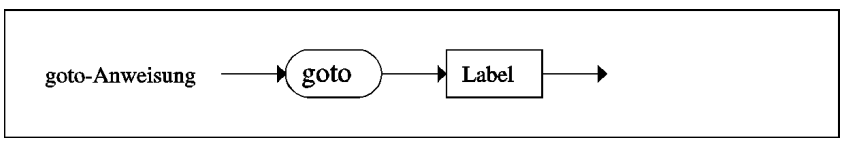


Abbildung 7.5 Syntax der goto-Anweisung

Bei der Verwendung von **goto** müssen die folgenden Regeln beachtet werden:

- Das Label, auf das **goto** Bezug nimmt, muß im selben Block stehen wie die **goto**-Anweisung selber. Anders gesagt: es ist nicht möglich, mit **goto** beliebig zwischen Prozeduren und/oder Funktionen hin- und herzuspringen.

■ Der Bezug auf einen strukturierten Anweisungsblock von einem Programmteil außerhalb dieses Blocks (d.h. ein Sprung auf eine tiefere Verschachtelungsebene) kann unvorhersehbare Folgen haben. Der Compiler erkennt derartige Fehler nicht!

Strukturierte Anweisungen

bestehen aus mehreren ineinander verschachtelten Ebenen, die ihrerseits Anweisungen enthalten. Sie werden entweder in der Reihenfolge ihres Erscheinens (Verbund- und **with**-Anweisungen), bedingt (konditionale Anweisung) oder wiederholt (Wiederholungsanweisungen bzw. Schleifen) ausgeführt.

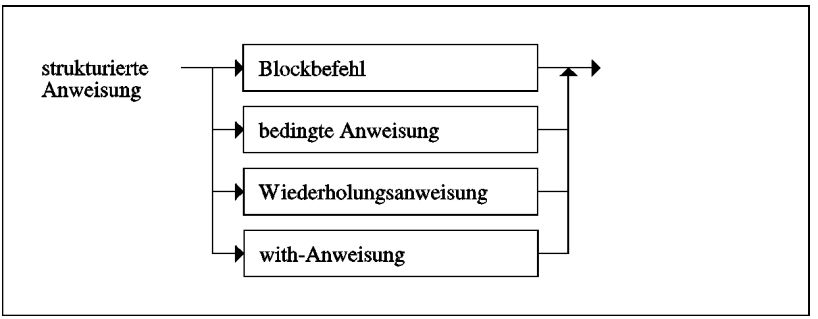


Abbildung 7.6 Syntax strukturierter Anweisungen

Verbundanweisungen

legen fest, daß die einzelnen darin enthaltenen Komponenten in der Reihenfolge ausgeführt werden, in der sie im Quelltext erscheinen. Alle im Verbund enthaltenen Anweisungen werden als ein einziger Block behandelt und genügen so den Forderungen an Stellen, wo die Syntax von Pascal nur eine einzelne Anweisung zuläßt. Anfang und Ende eines Verbundes werden durch `begin` und `end` gekennzeichnet, die einzelnen Komponenten sind durch Semikolons voneinander getrennt.

Ein Beispiel dazu:

```
var
  X, Y, Temp: Integer;
begin
   { ... }
   if X > Y then Writeln('X größer Y.')  { einfache Anweisung }
    else
    begin { Verbundanweisung }
```

```
        Temp := X; X := Y; Y:= Temp;
        Writeln('X und Y vertauscht.');
    end;
end.
```

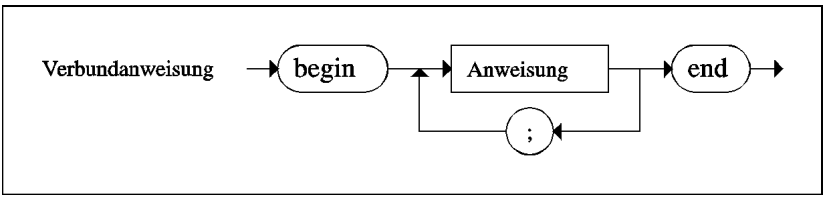


Abbildung 7.7 Syntax einer Verbundanweisung

Bedingte Anweisungen

bieten eine oder mehrere Optionen und wählen eine (gegebenenfalls auch keine) ihrer Komponenten zur Ausführung aus.

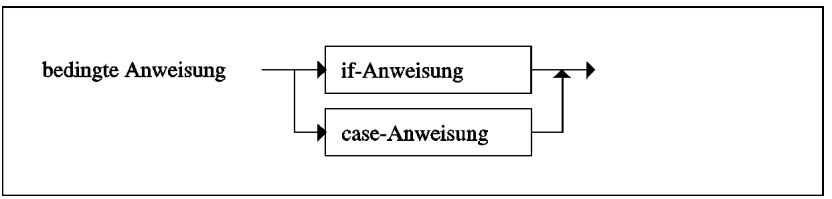


Abbildung 7.8 Syntax einer bedingten Anweisung

Die Anweisung if
läßt sich durch das folgende Syntaxdiagramm darstellen:

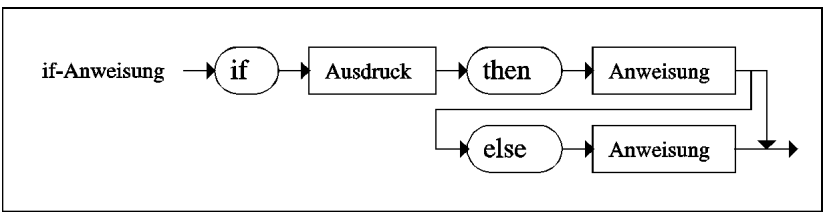


Abbildung 7.9 Syntax der if-Anweisung

Das Ergebnis eines solchen Ausdrucks muß den Standardtyp *Boolean* haben. Ergibt sich der Wert *True*, wird der auf **then** folgende Teil ausgeführt, ansonsten der auf **else** folgende Teil.

else ist optional: wenn dieser Zweig nicht vorhanden ist, bleibt die **if**-Anweisung »wirkungslos«, d.h. führt überhaupt nichts aus.

Einige Beispiele dazu:

```
if X < 1.5 then
  Z := X + Y
else
  Z :=1.5;

if P1 <> nil then
  Px := P1^.Root;
```

Mehrere aufeinanderfolgende Verschachtelungsebenen mit **if..then..else** sind syntaktisch doppeldeutig. Im folgenden Beispiel kann nicht eindeutig bestimmt werden, auf welches **if** sich das letzte **else** bezieht:

```
if e1 then if e2 then s1 else s2
```

Per Definition ist deshalb festgelegt, daß sich **else** auf das jeweils letzte **if** bezieht, zu dem noch kein **else**-Zweig angegeben wurde. Die zuvor gezeigte Folge wird also folgendermaßen interpretiert:

```
if e1 then
begin
  if e2 then
    s1
  else
    s2
end
```

Die Anweisung case
besteht aus einem Ausdruck (dem Selektor) und einer (beliebig langen) Liste von *Zweigen*. Jedem dieser Zweige gehen eine oder mehrere Konstanten (die **case**-Konstanten) oder das reservierte Wort **else** voraus. Der Selektor muß einen Ordinal-Typ haben, dessen obere und untere Grenzen im Bereich von -32768..65535 liegen – die Typen *LongInt* und *String* sind folglich als Selektor nicht erlaubt. Konstanten dürfen nicht mehrfach deklariert sein und müssen ebenfalls einen Ordinal-Typ haben, der darüber hinaus mit dem Typ des Selektors kompatibel ist.

Ein Zweig, dem eine **case**-Konstante vorausgeht, wird ausgeführt, wenn der Wert der Konstanten gleich dem des Selektors ist. Dasselbe gilt, wenn ein **case**-Bereich den Wert des Selektors enthält. Fällt der Wert des Selektors weder mit einer **case**-Konstanten noch einem **case**-Bereich zusammen, dann wird der auf das reservierte Wort **else** folgende Zweig ausgeführt. Wenn kein **else**-Zweig definiert ist, setzt Turbo Pascal die Programmausführung mit der nächsten Anweisung nach dem **case**-Block fort.

Auf jede **case**-Konstante kann eine einfache oder eine Verbundanweisung folgen. Das reservierte Wort **else** gilt innerhalb eines **case**-Blocks als Blockbeginn: zwischen **else** und dem abschließenden **end** ist deshalb eine beliebige Anzahl von Anweisungen möglich.

Einige Beispiele:

```
case Operator of
  plus:    X := X+Y;
```

```
    minus:   X := X-Y;
    mal:     begin X := X*Y; Y := 0 end
  end;

  case I of
    0,2,4,6,8 : Writeln('Gerade Zahl');
    1,3,5,7,9 : Writeln('Ungerade Zahl');
    10..100   : Writeln('Zwischen 10 und 100');
  else
     Writeln('Der Wert ist entweder negativ');
     Writeln(' oder größer als 100.');
```

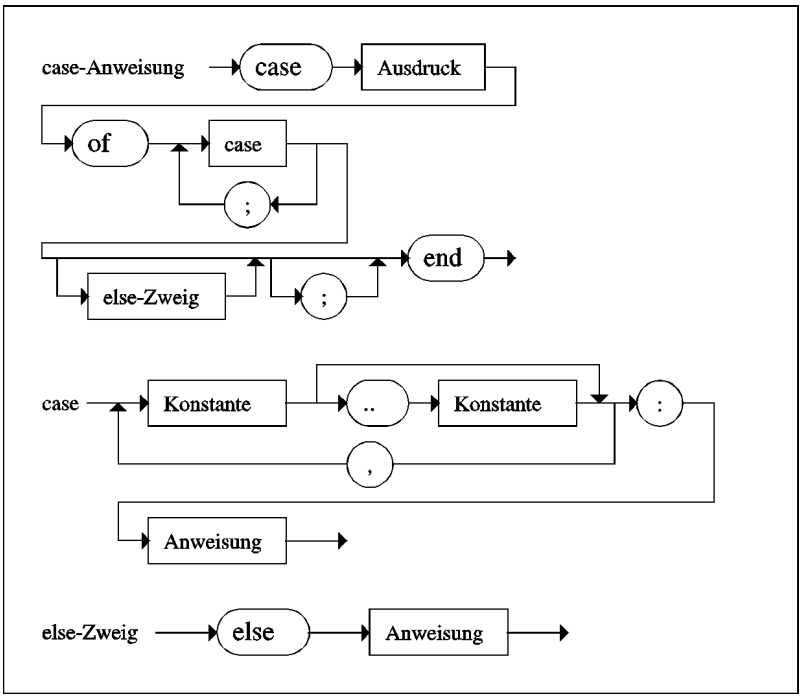


Abbildung 7.10 Syntax der case-Anweisung

```
  end;
```

Schleifen

Schleifen (oder Wiederholungsanweisungen) legen die wiederholte Ausführung bestimmter Programmteile fest.

Wenn die Anzahl der Wiederholungen von vornherein bekannt ist, sollte eine Konstruktion mit **for** verwendet werden – **repeat** und **while** sind haupt-

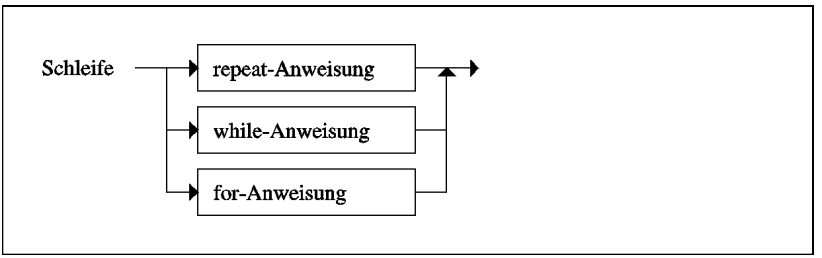


Abbildung 7.11 Syntax einer Schleifenanweisung

sächlich für Fälle gedacht, in denen die Abbruchbedingung durch Operationen innerhalb des Schleifenrumpfes gesetzt wird.

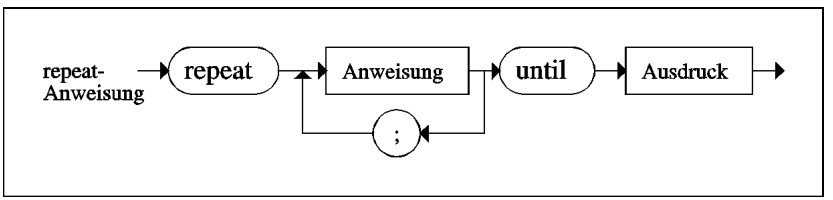


Abbildung 7.12 Syntax der repeat-Anweisung

Die repeat-Anweisung
enthält einen Ausdruck, dessen Wahrheitswert festlegt, ob der durch **repeat..-until** eingeschlossene Block wiederholt werden soll.

Das Ergebnis des Ausdrucks muß vom Typ *Boolean* sein. Die Anweisungen zwischen **repeat** und **until** werden solange nacheinander ausgeführt, bis der Ausdruck hinter **until** den Wert *True* annimmt. Die Anweisungsfolge wird mindestens einmal ausgeführt, weil der Ausdruck erst nach Erreichen von **until** bewertet wird. Zwei Beispiele zu **repeat**:

```
repeat
  K := I mod J;
  I := J;
  J := K;
until J = 0;
```

```
repeat
  Write('Geben Sie einen Wert von 0 bis 9 ein:  ');
  Readln(I);
until (I >= 0) and (I <= 9);
```

Die while-Anweisung
enthält einen Ausdruck, dessen Wahrheitswert bestimmt, ob die auf **while** folgende einfache bzw. Verbundanweisung wiederholt ausgeführt wird oder nicht.

Wie bei **repeat** muß der Ausdruck den Typ *Boolean* haben. Er wird jeweils *vor*

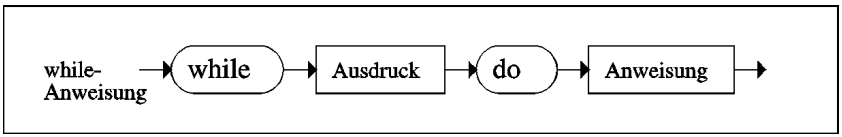


Abbildung 7.13 Syntax der while-Anweisung

der Ausführung des zu wiederholenden Programmteils bewertet – wenn sich bereits vor der ersten Ausführung der Wert *False* ergibt, wird der auf **while** folgende Block überhaupt nicht ausgeführt. Ansonsten wird er solange wiederholt, bis der Ausdruck den Wert *False* annimmt.

Der wesentliche Unterschied zwischen **while** und **repeat** liegt also darin, daß **while** eine Schleife wiederholt, *solange* der Ausdruck *True* ergibt; **repeat** wiederholt die Schleife, *bis* der Ausdruck den Wert *True* annimmt.

Einige Beispiele zu **while**:

```
while Daten[I] <> x do I := I + 1;

while I > 0 do
begin
  if Odd(I) then Z := Z * X;
  I := I div 2;
  X := Sqr(X);
end;

while not Eof(Eingabedatei) do
begin
  Readln(Eingabedatei, Zeile);
  Bearbeiten(Zeile);
end;
```

Die for-Anweisung

führt eine einfache oder eine Verbundanweisung in einer Schleife aus, wobei einer Variablen (der Laufvariablen) fortlaufende Werte zugewiesen werden.

Die Laufvariable muß der Bezeichner einer Variablen ordinalen Typs sein, die entweder innerhalb desselben Blocks wie die **for**-Anweisung lokal oder global zum gesamten Programm deklariert ist. Die Definition einer Schleife mit **for** schließt die Festlegung eines Start- und Endwertes mit ein. Beide Werte müssen einen Ordinal-Typ haben, der zu dem der Laufvariablen zuweisungskompatibel ist.

Beim Start der Schleife wird die Laufvariable auf den Startwert gesetzt und für jeden Schleifendurchlauf um eins erhöht bzw. erniedrigt – solange, bis der Endwert erreicht ist. Bei jedem Durchlauf wird die im Rumpf der Schleife enthaltene Anweisung bzw. Verbundanweisung einmal ausgeführt. Wenn die Endbedingung der Schleife bereits vor dem ersten Durchlauf gegeben ist (d.h. Endwert < Startwert bzw. Endwert > Startwert bei Verwendung von **downto**), dann werden Schleife und dazugehöriger Rumpf komplett übersprungen.

Zuweisungen an die Laufvariable innerhalb einer **for**-Schleife sind unzu-

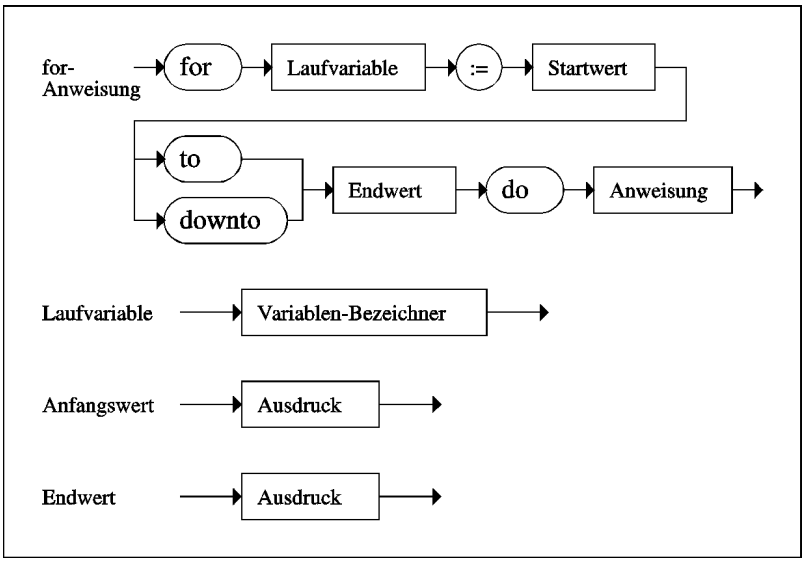


Abbildung 7.14 Syntax der for-Anweisung

lässig. Der Wert der Laufvariablen nach Ende der Schleife ist undefiniert – es sei denn, die Ausführung der Schleife wurde durch ein **goto** innerhalb des Rumpfes abgebrochen. Unter Beachtung dieser Einschränkungen führt die Anweisung

```
for V := Expr1 to Expr2 do
     Rumpf;
```

zum selben Ergebnis wie das Konstrukt:

```
begin
  Temp1 := Expr1;
  Temp2 := Expr2;
   if Temp1 <= Temp2 then
  begin
    V := Temp1;
    Rumpf;
    while V <> Temp2 do
    begin
       V := Succ(V);
       Rumpf;
    end;
  end;
end;
```

Analoges gilt für **for**-Anweisungen mit **downto** – auch hier sind die beiden folgenden Konstrukte praktisch identisch:

```
for V := Expr1 downto Expr2 do
       Rumpf;
```

```
begin
  Temp1 := Expr1;
  Temp2 := Expr2;
  if Temp1 >= Temp2 then
  begin
    V := Temp1;
    Rumpf;
    while V <> Temp2 do
    begin
      V  := Pred(V);
      Rumpf;
    end;
  end;
end;
```

Temp1 und *Temp2* sind hier Hilfsvariablen des Grundtyps der Variablen *V* und erscheinen an keiner anderen Stelle im Programm. Einige weitere Beispiele für **for**-Anweisungen:

```
for I := 2 to 63 do
  if Daten[I] > Maximum then Maximum := Daten[I];
```

```
for I := 1 to 10 do
  for J := 1 to 10 do
    begin
      X := 0;
      for K := 1 to 10 do
           X := X + Mat1[I,K] * Mat2[K,J];
      Mat[I,J] := X;
  end;
```

```
for C := rot to blau do Farbwahl(C);
```

Die with-Anweisung

verkürzt den Bezug auf Felder von Records sowie auf Komponenten von Objekten (Felder, Methoden, Konstruktoren und Destruktoren). Zur Vereinfachung ist im folgenden Text ausschließlich von Record-Feldern die Rede, obgleich sowohl Record-Felder als auch Objekt-Komponenten gemeint sind.

Innerhalb einer **with**-Anweisung kann auf die Felder einer oder mehrerer Record-Variablen Bezug genommen werden, wobei lediglich die Angabe der Feldbezeichner notwendig ist.

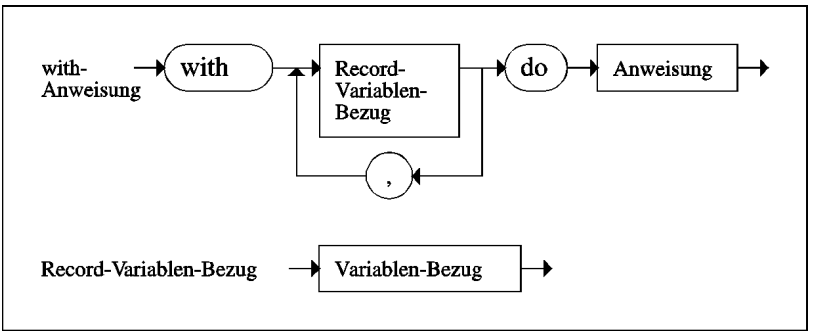


Abbildung 7.15 Syntax der with-Anweisung

Die beiden folgenden Beispiele sind also identisch:

```
with Datum do
  if Monat = 12 then
  begin
    Monat := 1;
    Jahr := Jahr + 1;
  end else
    Monat := Monat + 1;
```

```
if Datum.Monat = 12 then
begin
  Datum.Monat := 1;
  Datum.Jahr  := Datum.Jahr + 1;
end else
  Datum.Monat := Datum.Monat + 1;
```

Innerhalb des **with**-Blocks wird jeder Variablenbezug daraufhin geprüft, ob er als Feld eines Records behandelt werden kann. Fällt diese Prüfung positiv aus, wird die entsprechende Variable immer als Feld interpretiert – selbst dann, wenn auch auf eine andere Variable gleichen Namens zugegriffen werden könnte. Das folgende Beispiel verdeutlicht diese Problematik. Nach einer Deklaration wie

```
type
  Point = record
        x,y: Integer;
  end;

var
  x: Point;
  y: Word;
```

können sich *x* und *y* – je nach Kontext – entweder auf Variablen oder Felder des Records beziehen, also:

```
with x do
   begin
```

```
      x := 10;    {Feld x des Records}
      y := 20;    {Feld y des Records}
    end;
y := 20;     {Word-Variable y}
```

with-Anweisungen können mehrfach verschachtelt werden. Die folgenden Zeilen sind identisch:

```
with V1,V2,...Vn do <Anweisung>;
```

```
with V1 do
  with V2 do
    ....
      with Vn do
        <Anweisung>;
```

Wenn *Vn* ein Feld von *V1 und* von *V2* ist, wird es in beiden Fällen als *V2.Vn* und nicht als *V1.Vn* interpretiert. Wenn die Adressierung einer Record-Variablen die Indizierung eines Arrays oder die Ermittlung einer Zeiger-Adresse erfordert, wird die entsprechende Aktion vor dem Beginn des Anweisungsteils ausgeführt.

8. Prozeduren und Funktionen

Prozeduren und Funktionen stellen formal gesehen zusätzliche Ebenen innerhalb des Hauptprogramm-Blocks dar, also eine Verschachtelung. Eine Prozedur wird durch einen Prozeduraufruf (d.h. die Angabe ihres Bezeichners) aktiviert und liefert keinen direkten Wert zurück. Eine Funktion wird bei der Berechnung eines Ausdrucks aktiviert, in der ihr Bezeichner erscheint, und hat normalerweise ein Ergebnis, das für diesen Aufruf mit dem Funktionsbezeichner gleichgesetzt werden kann. Dieses Kapitel erläutert die verschiedenen Typen von Prozedur- und Funktionsdeklarationen sowie dazugehörige Parameter.

Prozedur-Deklarationen

Eine Deklaration, die mit dem reservierten Wort **procedure** eingeleitet wird, verbindet einen Bezeichner und einen Block von Anweisungen zu einer Prozedur. Derartig deklarierte Prozeduren können durch Angabe ihres Bezeichners aktiviert (d.h. aufgerufen) werden.

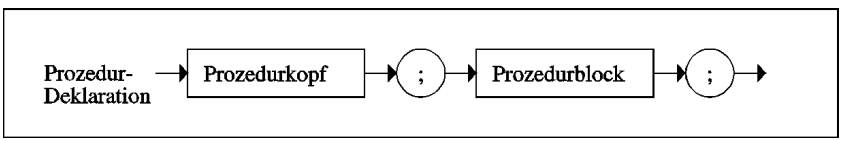


Abbildung 8.1 Syntax einer Prozedur-Deklaration

Der Prozedurkopf benennt die Prozedur (d.h. ordnet ihr einen Bezeichner zu) und deklariert formale Parameter (soweit vorhanden).

Die Syntax einer formalen Parameterliste wird im Abschnitt »Parameter« in diesem Kapitel besprochen. Durch die Angabe ihres Bezeichners sowie einer Liste aktueller Parameter wird eine Prozedur *aktiviert*: Die im Anweisungsteil der entsprechenden Prozedurdeklaration definierten Aktionen werden ausgeführt.

Eine Prozedur, die ihre eigene Aktivierung als Bestandteil ihres Anweisungsteils enthält, wird rekursiv ausgeführt, d.h. sie ruft sich selber wiederholt auf.

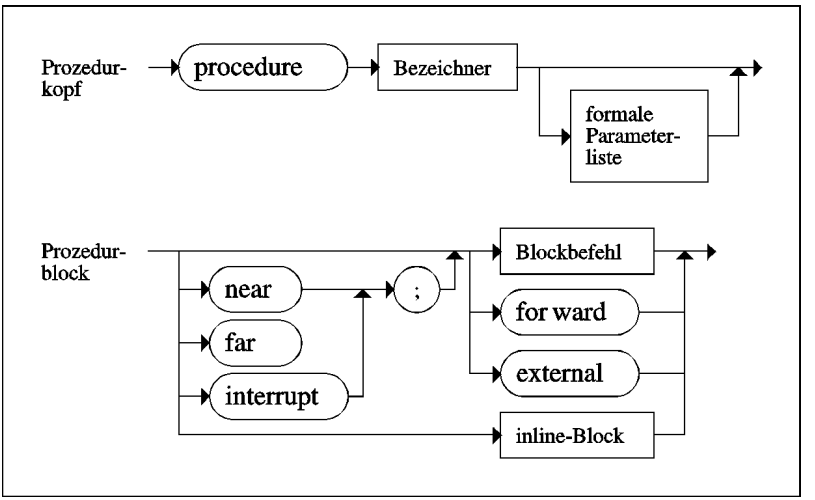


Abbildung 8.2 Syntax eines Prozedurkopfes

Ein Beispiel für eine Prozedurdeklaration:

```
procedure NumString(N:Integer; var S: string);
var
  v: Integer;    {lokale Variable}
begin
  V := Abs(N);
  S := '';
  repeat
    S := Chr(N mod 10 + Ord('0')) + S;
    N := N div 10;
  until N = 0;
  if N < 0 then S := '-'+ S;
end;
```

Near- und far-Deklarationen

Turbo Pascal unterstützt zwei Modelle von Prozedur-Aufrufen: **near** und **far**. Bezüglich Codegröße und Ausführungsgeschwindigkeit ist das **near**-Modell dem **far**-Modell überlegen, aber es besitzt die Einschränkung, daß die Prozeduren nur von dem Modul aus aufgerufen werden können, in dem sie deklariert wurden. **far**-Prozeduren hingegen können von überall angesprungen werden, aber der Code für **far**-Aufrufe ist etwas langsamer und größer als der von **near**-Aufrufen.

Aufgrund der Prozedur-Deklaration wählt Turbo Pascal automatisch das richtige Aufruf-Modell aus: Prozeduren, die im **interface**-Teil einer Unit deklariert wurden, benutzen das **far**-Modell. Prozeduren, die nur im **implemen-**

tation-Teil einer Unit auftauchen, erhalten das **near**-Modell. Sie können ohnehin nur innerhalb der Unit bzw. des Programmes aufgerufen werden.

In einigen speziellen Fällen werden Prozeduren benötigt, die das **far**-Modell benutzen: Zum Beispiel können nur **far**-Prozeduren einer Prozedur-Variablen zugewiesen werden. Die Compiler-Direktive `$F` kann eingesetzt werden, um die automatische Modell-Auswahl des Compilers zu überschreiben.

Prozeduren und Funktionen, die im `{$F+}`-Modus übersetzt wurden, benutzen immer das **far**-Modell. Im voreingestellten `{$F-}`-Modus wählt der Compiler automatisch das richtige Aufruf-Modell.

Um für eine Prozedur oder Funktion ein ganz bestimmtes Aufruf-Modell zu setzen, kann vor dem Prozedur-Block eine far- bzw. near-Direktive geschrieben werden. Eine solche Direktive überschreibt sowohl die `$F`-Direktive als auch die automatische Wahl des Aufruf-Modells.

Deklarationen mit export

Die Direktive **export** macht eine Prozedur oder Funktion »exportierbar«. Eine solche Routine wird automatisch als **far** codiert, und der Compiler generiert speziellen Eintritts- und Austrittcode.

Unter Windows müssen Prozeduren und Funktionen in folgenden Fällen exportierbar sein:

- die Prozedur oder Funktion steht innerhalb einer dynamischen Linkbibliothek (DLL) zur Verfügung;
- die Routine ist eine Callback-Prozedur oder -Funktion.

In Kapitel 10 »Dynamische Linkbibliotheken« wird die Deklaration exportierbarer Prozeduren und Funktionen innerhalb einer DLL eingeheneder behandelt. Beachten Sie aber, daß eine solche Routine erst dann tatsächlich exportierfähig ist, wenn sie auch in der der Anweisung **exports** folgenden Liste aufgeführt ist.

Callback-Prozeduren und -Funktionen sind in Ihrem Programm definierte Routinen, die nicht von diesem selbst, sondern von Windows aufgerufen werden. Solche Routinen müssen die Direktive **export** aufweisen, brauchen aber nicht in der **exports**-Liste aufgeführt zu werden.

Deklarationen mit interrupt

In Turbo Pascal kann eine Prozedurdeklaration mit dem ebenfalls optionalen Zusatz **interrupt** versehen werden, der direkt auf den Prozedurkopf folgt. Eine vollständige Besprechung derartiger Prozeduren finden Sie in Kapitel 14 »Interne Details« – wir wollen an dieser Stelle nur auf die formalen Voraussetzungen eingehen:

- **interrupt**-Prozeduren können nicht wie andere Prozeduren über ihren Namen aufgerufen werden, sondern nur über Interrupt-Vektoren des Systems.
- Der Kopf einer als **interrupt** deklarierten Prozedur hat ein fixes Format mit einer festgelegten Anzahl und Reihenfolge von Parametern, nämlich:

```
procedure <Name> (Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,ES,BP: Word);
interrupt;
```

Auf den Kopf einer »normalen« Prozedur kann anstelle eines Blocks auch eine der Direktiven forward, external oder inline folgen. Zweck und Wirkung dieser Deklarationen werden in den folgenden Abschnitten besprochen.

Deklarationen mit forward

Eine Prozedurdeklaration, auf die anstelle eines Vereinbarungs- und Anweisungsteils das reservierte Wort **forward** folgt, wird als **forward**-Deklaration bezeichnet. Der eigentliche Prozedurblock muß zu einem späteren Zeitpunkt (d.h. weiter »unten« im Programmtext) definiert werden. Diese Definition beginnt wiederum mit **procedure**, gefolgt vom Bezeichner der als **forward** deklarierten Prozedur, enthält jedoch keine formale Parameterliste.

Zwischen der **forward**-Deklaration und der Definition der Prozedur ist die Deklaration weiterer Prozeduren und Funktionen möglich, die ihrerseits die als **forward** deklarierte Prozedur aufrufen können. Auf diese Weise ist der sich überkreuzende Bezug zwischen zwei oder mehreren Prozeduren herstellbar (d.h. die Definition von Programmteilen, die sich gegenseitig aufrufen).

Hinweis: Die **forward**-Deklaration und die Definition der Prozedur müssen in ein und demselben Deklarationsteil des Programms stehen. In einem Konstrukt wie

```
procedure FF; forward;    { forward-Deklaration }
....
type                      { eingeschobener Typen-Definitionsteil }
  ....
procedure FF;             { Definition der Prozedur }
begin
  ...
```

wird der durchgehende Deklarationsteil des Programms von einem Typen-Definitionsteil unterbrochen – folglich meldet der Compiler einen Fehler. Analoges gilt für »Unterbrechungen« durch die Deklaration globaler Variablen und/oder Konstanten.

Die **forward**-Deklaration bildet zusammen mit dem Prozedurblock eine vollständige Prozedur. Die Prozedur selber gilt mit **forward** als deklariert. Ein Beispiel:

```
procedure Walter(m,n: Integer); forward;

rocedure Klara(x,y: Real);
```

```
begin
   ....
  Walter(4,5);
   ....
end;

procedure Walter;            {Definition der forward-Prozedur}
begin
   ....
  Klara(8.3,2.4);
   ....
end;
```

Die Definition einer als **forward** deklarierten Prozedur kann als **external** oder als **assembler** geschehen – **far, near, inline, interrupt** oder die Definition über ein weiteres **forward** sind dagegen nicht zulässig.

Im Interface-Teil einer Unit sind **forward**-Deklarationen weder notwendig noch erlaubt.

Deklarationen mit external

ermöglichen die Einbindung (das »Linken«) getrennt compilierter Prozeduren und Funktionen, die in Maschinensprache geschrieben worden sind. Der externe Code wird über den Compiler-Befehl {$L Dateiname} mit einem Pascal-Programm oder einer Unit verbunden.

Einige Beispiele dazu:

```
procedure MoveWord(var Qu, Zi; Zahl: LongInt); external;
procedure MoveLong(var Qu, Zi; Zahl: LongInt); external;
procedure FillW(var Ziel; Data: Integer; Z:LongInt); external;
procedure FillL(var Ziel; Data: LongInt; Z:LongInt); external;

$L Blkstuff.obj}    {<- eine selbstgeschriebene Datei}
```

Weitere Details zu **external**-Prozeduren finden sich in Kapitel 23 »Einbinden von Assembler-Code«. Darüber hinaus dient die **external**-Direktive dazu, Prozeduren und Funktionen aus einer dynamischen Linkbibliothek zu *importieren*. Das folgende Beispiel importiert die Funktion *GlobalAlloc* aus der zu Windows gehörenden DLL *KERNEL*:

```
function GlobalAlloc(Flags: Word; Bytes: LongInt): THandle; far;
external 'KERNEL' index 15;
```

Die **external**-Direktive ersetzt den Deklarations- und Anweisungsteil einer importierten Prozedur oder Funktion. Diese muß als **far** deklariert sein, was sowohl über den Compiler-Befehl **{$F+}** als auch über die **far**-Deklaration möglich ist. Abgesehen von diesen Voraussetzungen entspricht eine importierte völlig einer »normalen« Prozedur oder Funktion.

Deklarationen mit assembler

Mit Hilfe einer **assembler**-Deklaration können ganze Prozeduren und Funktionen mit dem integrierten Assembler geschrieben werden.

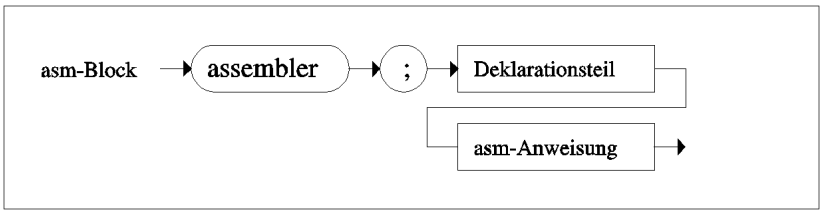


Abbildung 8.3 Syntax einer assembler-*Deklaration*

In Kapitel 22 »Der integrierte Assembler« finden Sie weitere Informationen zu **assembler**-Deklarationen.

Deklarationen mit inline

machen es möglich, anstelle eines Pascal-Anweisungsblocks Maschinencode zu verwenden: Nach dem reservierten Wort **inline** erwartet der Compiler eine Liste von Konstanten (normalerweise in hexadezimaler Notation), die direkt in das Programm aufgenommen werden.

Im Normalfall generiert der Compiler bei einem Prozeduraufruf Code, der die Argumente der Prozedur auf den Stack legt, danach erzeugt er einen CALL-Befehl zum Aufruf der Prozedur. Wenn eine **inline**-Prozedur »aufgerufen« wird, werden die Konstanten, die nach **inline** definiert sind, vom Compiler ohne einen CALL direkt in den laufenden Code eingesetzt. Jede Konstante bildet genau ein Wort im generierten Code, der Compiler bildet bei der Codegenerierung die Reihenfolge der Konstanten ab.

inline sollte anstelle von **external** zur Implementation kleinerer Routinen (mit maximal 10 oder 20 Prozessorbefehlen) verwendet werden. Zwei Beispiele für **inline**-Prozeduren:

```
procedure Interrupts_An; inline ($FA);  { CLI }
procedure Interrupts_Aus; inline ($FB); { STI }
```

Eine vollständige Besprechung von **inline**-Prozeduren finden Sie in Kapitel 23 »Einbinden von Assembler-Code«.

Funktions-Deklarationen

Eine Funktions-Deklaration definiert einen Programmteil, der einen Wert berechnet und als Funktionsergebnis zurückliefert.

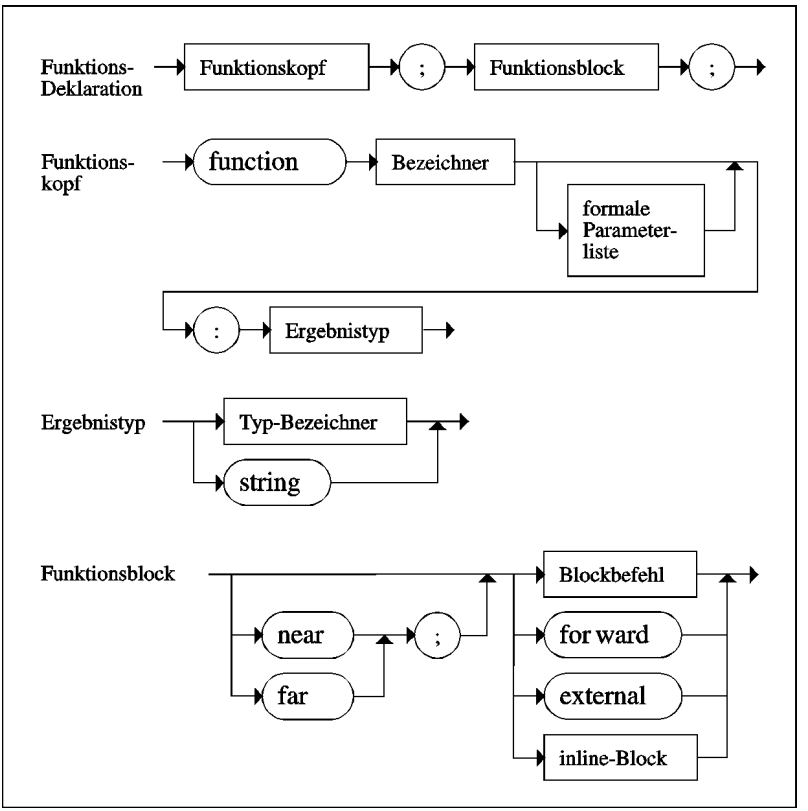


Abbildung 8.4 Syntax einer Funktion

Der Funktionskopf legt den Bezeichner, die formalen Parameter (soweit vorhanden) und den Ergebnistyp der Funktion fest:

Funktionen werden bei der Berechnung von Ausdrücken aktiviert, in denen ihr Bezeichner erscheint, und stehen dort stellvertretend für den von ihnen zurückgelieferten Wert. Ein Funktionsbezeichner kann überall anstelle eines Operanden in einem Ausdruck eingesetzt werden, solange der Typ des Funktionsergebnisses mit dem des ersetzten Operanden kompatibel ist. Zuweisungen an einen Funktionsbezeichner sind nur innerhalb der Funktion selber erlaubt.

Der Aufruf einer Funktion geschieht über die Angabe ihres Bezeichners, gefolgt von einer Liste aktueller Parameter, die in Typ und Reihenfolge den formalen Parametern der Funktionsdeklaration entsprechen müssen.

Der Anweisungsblock einer Funktion legt alle Operationen fest, die nach der Aktivierung der Funktion ausgeführt werden sollen. Dieser Block sollte zumindest eine Zuweisung enthalten, die dem Funktionsbezeichner einen Wert gibt. Das Ergebnis der Funktion ist der jeweils letzte zugewiesene Wert vor dem Ende des Anweisungsteils der Funktion. Existiert eine solche Zuweisung nicht, oder wird sie aus irgendeinem Grund nicht ausgeführt, bleibt der Wert der Funktion undefiniert.

Wenn eine Funktion ihren eigenen Bezeichner innerhalb des Anweisungsteils in einem Ausdruck aufführt, ist das Ergebnis ein rekursiver Aufruf – die Funktion ruft sich also selbst erneut auf. Einige Beispiele zu Funktionsdeklarationen:

```
function Max(a: Vektor; n: Integer): Extended;
var
  x: Extended;
  i: Integer;
begin
  x := a[1];
  for i := 2 to n do
    If x < a[i] then x := a[i];
  Max := x;
end;

function Potenz(x: Extended; y: Integer): Extended;
var
  z: Extended;
  i: Integer;
begin
  z := 1.0; i:= y;
  while i > 0 do begin
    If Odd(i) then z := z * x;
    i := i div 2;
    x := Sqr(x);
   end;
  Potenz := z;
end;
```

Funktionen können in derselben Weise wie Prozeduren als **forward, export, external, near, far, assembler** oder **inline** deklariert werden – der Zusatz **interrupt** ist dagegen nicht erlaubt.

Deklarationen von Methoden

Die Deklaration einer Methode innerhalb eines Objekttyps entspricht einer **forward**-Deklaration dieser Anweisung. Deshalb muß die Methode nach der

Deklaration des Objekttyps im selben Gültigkeitsbereich mit Hilfe einer Definition *implementiert* werden.

Werden Prozeduren und Funktionen als Methoden definiert, so hat diese Definition die Form einer normalen Prozedur- oder Funktionsdeklaration, allerdings mit der Ausnahme, daß in diesem Fall als Prozedur- oder Funktionsbezeichner ein vollständiger Methodenbezeichner verwendet werden muß.

Die Definition von Konstruktoren und Destruktoren hat dieselbe Form wie die Definition einer Methode, die als Prozedur deklariert wird, außer, daß das Schlüsselwort **procedure** durch die reservierten Wörter **constructor** und **destructor** ersetzt wird.

Die Definition einer Methode kann optional die formale Parameterliste des Deklarationskopfs der Methode im Objekttyp wiederholen. Die Kopfzeile der Definition muß in diesem Fall in Bezug auf Reihenfolge, Typ und Bezeichner sowie gegebenenfalls dem Funktionsresultat exakt übereinstimmen.

In der Definition einer Methode ist immer automatisch die Bezeichnung *Self* deklariert, die formal einem Variablenparameter vom Typ des jeweiligen Objekts entspricht. Innerhalb des Anweisungsblocks stellt *Self* die Instanz dar, über deren Methode die Methode aktiviert wurde. Deshalb wirken sich alle Veränderungen der Felder von *Self* auch in der Instanz selbst aus.

Der Gültigkeitsbereich einer Komponentenbezeichnung in einem Objekttyp reicht über alle Anweisungsblöcke von Prozeduren, Funktionen, Konstruktoren und Destruktoren, in denen eine Methode des Objekttyps implementiert ist. Das bedeutet dasselbe, als stünde der gesamte Methodenblock in einer **with**-Anweisung der Form

```
with Self do begin ... end
```

Aus diesem Grund muß die Schreibweise von Komponentenbezeichnungen, formalen Parametern, Self und allen lokalen Variablen in einer Methode eindeutig sein. Nachfolgend sehen Sie nun einige Beispiele für die Implementierung von Methoden:

```
procedure Rect.Intersect(var R : Rect);
begin
  if A.X < R.A.X then A.X := R.A.X;
  if A.Y < R.A.Y then A.Y := R.A.Y;
  if B.X > R.B.X then B.X := R.B.X;
  if B.Y > R.B.Y then B.Y := R.B.Y;
  if (A.X >= B.X) or (A.Y >= B.Y) then Init(0, 0, 0, 0);
end;

procedure Field.Display;
begin
  GotoXY(X, Y);
  Write(Name^, ' ', GetStr);
end;

function NumField.PutStr(S : String) : Boolean;
var
  E: Integer;
begin
```

```
  Val(S, Wert, E);
  PutStr := (E = 0) and (Value >= Min) and (Value <= Max);
end;
```

Konstruktoren und Destruktoren

Konstruktoren und Destruktoren sind zwei spezielle Formen von Methoden. In Verbindung mit der erweiterten Syntax der Standardprozeduren *New* und *Dispose* kann man mit einem Konstruktor dynamische Objekte im Speicher anlegen und mit einem Destruktor den durch die Objekte belegten Speicherplatz wieder freigeben. Zusätzlich bieten Konstruktoren die Möglichkeit, die erforderliche Initialisierung der Objekte, in denen virtuelle Methoden enthalten sind, vorzunehmen. Wie andere Methoden können Konstruktoren und Destruktoren vererbt werden. Ein Objekt kann beliebig viele Konstruktoren und Destruktoren enthalten.

Konstruktoren werden zur Initialisierung von Instanzen eines Objekts verwendet. Normalerweise funktioniert die Initialisierung so, daß die Initialisierungswerte als Parameter an den Konstruktor übergeben werden. Konstruktoren dürfen nicht virtuell sein, weil der Mechanismus zum Ermitteln der Adresse einer virtuellen Methode in der VMT von einem Konstruktor abhängt, mit dem das Objekt vorher bereits initialisiert worden sein muß.

Hier sehen Sie nun einige Beispiele für Konstruktoren:

```
constructor Field.Copy(var F :Field);
begin
  Self := F;
end;

constructor Field.Init(FX, FY, FLen : Integer; FName : String);
begin
  X := FX;
  Y := FY;
  Len := FLen;
  GetMem(Name, Length(FName) + 1);
  Name^ := FName;
end;

constructor StrField.Init(FX, FY, FLen : Integer; FName : String);
begin
  Field.Init(FX, FY, FLen, FName);
  GetMem(Wert, FLen);
  Value^ := '';
end;
```

Die erste Aktion des Konstruktors eines Nachkommens, wie zum Beispiel des oben dargestellten *StrField.Init*, ist fast immer der Aufruf des entsprechenden Konstruktors des direkten Vorfahren, um die übernommenen Felder des Objekts

zu initialisieren. Nachdem das geschehen ist, initialisiert der Konstruktor die Felder des Objekts, die vom Nachkommen eingeführt wurden.

Destruktoren sind die Gegenstücke zu den Konstruktoren und werden zum Aufräumen der Objekte nach erfolgter Anwendung verwendet. Normalerweise besteht das Aufräumen in der Freigabe aller Zeigerfelder in einem Objekt.

Im Gegensatz zu Konstruktoren können Destruktoren virtuell sein. Sie besitzen üblicherweise keine formalen Parameter.

Hier sehen Sie einige Beispiele für Destruktoren:

```
destructor Field.Done;
begin
  FreeMem(Name, Length(Name^) + 1);
end;

destructor StrField.Done;
begin
  FreeMem(Value, Len);
  Field.Done;
end;
```

Ein Destruktor eines Nachkommens, wie zum Beispiel des oben aufgeführten *StrField.Done*, gibt normalerweise zuerst die vom Nachkommen eingeführten Zeigerfelder frei und ruft dann als letzte Aktion den entsprechenden Destruktor des direkten Vorgängers auf, um alle übernommenen Zeigerfelder des Objekts aus dem Speicher zu entfernen.

Parameter

Die Deklaration einer Prozedur oder Funktion schließt eine Liste formaler Parameter mit ein. Jeder formal deklarierte Parameter ist lokal zur entsprechenden Funktion oder Prozedur, innerhalb des Blocks kann über die Angabe seines formalen Namens auf ihn Bezug genommen werden.

Turbo Pascal unterscheidet drei Arten von Parametern:

- Parameter, denen nicht das reservierte Wort **var** vorausgeht, werden als Wertparameter behandelt;
- Parameter, denen das reservierte Wort **var** vorausgeht und eine Typ-Bezeichnung folgt, werden als Variablenparameter behandelt;
- Parameter, denen das reservierte Wort **var** vorausgeht, aber keine Typ-Bezeichnung folgt, werden als untypisierte Parameter behandelt.

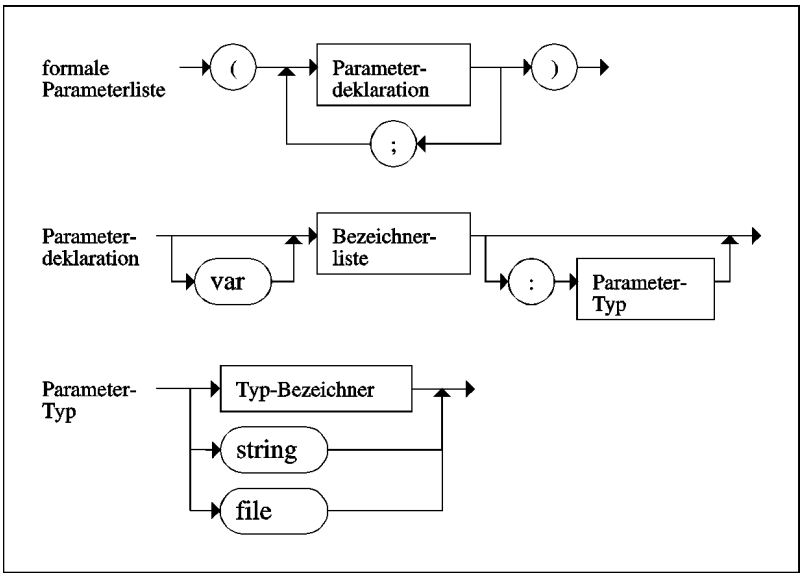


Abbildung 8.5 Syntax von Parametern

Wert-parameter

Ein formaler Wertparameter verhält sich wie eine lokale Variable der entsprechenden Prozedur bzw. Funktion – der einzige Unterschied besteht darin, daß er beim Aufruf den Wert eines aktuellen Parameters erhält. Veränderungen von Wertparametern beeinflussen den Wert des aktuellen Parameters nicht.

Der korrespondierende aktuelle Parameter eines formalen Parameters kann – im Gegensatz zu Variablenparametern – ein Ausdruck sein. Der Wert dieses Ausdrucks darf weder einen Dateityp noch einen strukturierten Typ haben, der einen Dateityp als Komponente enthält.

Der aktuelle Parameter muß zu seinem formalen Parameter zuweisungskompatibel sein. Wenn der Parameter den Typ **string** hat, wird seine Maximallänge automatisch auf 255 Zeichen festgelegt.

Variablen-parameter

Ein Variablenparameter wird benutzt, wenn ein Wert einer Prozedur oder Funktion durch Bezugnahme übergeben werden soll. Der korrespondierende aktuelle Parameter innerhalb eines Prozedur- oder Funktionsaufrufs muß ein Variablenbezug sein – die Übergabe von Ausdrücken als Variablenparameter ist also nicht zulässig. Der formale Variablenparameter repräsentiert während der Aktivierung der Prozedur oder Funktion den aktuellen Parameter. Jede Verän-

derung des formalen Variablenparameters beeinflußt auch den Wert des aktuellen Parameters (d.h. der übergebenen Variablen).

Innerhalb der Prozedur oder Funktion wird mit jedem Bezug auf den formalen Parameter auf den Wert des aktuellen Parameters zugegriffen. Beide Parameter müssen denselben Typ haben – eine Einschränkung, die durch die Verwendung untypisierter Parameter umgangen werden kann. Hat der formale Parameter den Typ **string**, erhält er das Längenattribut 255 – der aktuelle Variablenparameter muß in diesem Fall dasselbe Längenattribut besitzen.

Dateitypen können ausschließlich als Variablenparameter übergeben werden. Operationen wie die Indizierung eines Arrays oder die Berechnung einer Zeigeradresse für den Bezug auf aktuelle Variablenparameter werden vor der Übergabe des Parameters ausgeführt.

Für Variablenparameter mit Objekttypen gelten die Regeln für die Kompatibilität von Zuweisungen zwischen Objekten (vgl. Kapitel 7 »Anweisungen«). Für einen formalen Parameter des Typs *T1* kann der aktuelle Parameter den Typ *T2* haben, wenn *T2* im Bereich von *T1* liegt. Zum Beispiel kann der Methode *Field.Copy* eine Instanz von *Field*, *StrField*, *NumField*, *ZipField* oder eine andere Instanz eines Nachkommens von Field übergeben werden.

Untypisierte Variablenparameter

Wenn der Typ eines formalen Parameters nicht definiert ist, d.h. dem dazugehörigen Bezeichner im Prozedur- bzw. Funktionskopf kein Typ-Bezeichner folgt, dann wird dieser Parameter als untypisiert bezeichnet. In diesem Fall kann der entsprechende aktuelle Parameter jeden beliebigen Typ haben.

Untypisierte Variablen sind inkompatibel zu allen anderen Variablen – schließlich haben sie keinen Typ. Deshalb muß ihnen vor ihrer Verwendung über eine Typumwandlung ein spezifischer Typ zugewiesen werden. Ein Beispiel für eine Funktion, die mit zwei untypisierten Parametern arbeitet:

```
function Gleich(var Quelle, Ziel; Zahl : Integer): Boolean
type
  Bytes = array[0..MaxInt] of Byte;
var
  N: Integer;
begin
  N:= 0;
  while (N < Zahl) and (Bytes(Quelle)[N] <> Bytes(Ziel)[N]) do
    Inc(N);
  Gleich := N = Zahl;
end;
```

Mit dieser Funktion können zwei beliebige Variablen jeder Größe miteinander verglichen werden. Gegeben seien die folgenden Deklarationen und Funktionsaufrufe:

```
type
  Vector = array[1..10] of Integer;
  Point  = record
    x,y: Integer;
  end;

var
  Vec1, Vec2: Vector;
  N: Integer;
  P: Point;
  B: Boolean;

begin
  B := Gleich(Vec1,Vec2,SizeOf(Vector));
  Writeln(B);
  B := Gleich(Vec1,Vec2,Sizeof(Integer)*N);
  Writeln(B);
  B := Gleich(Vec[1],Vec1[6],SizeOf(Integer)*5);
  Writeln(B);
  B := Gleich(Vec1[1],P,4); Writeln(B);
end.
```

Der erste Aufruf vergleicht sämtliche Elemente von *Vec1* mit denen von *Vec2*; der zweite vergleicht nur die ersten N Komponenten beider Arrays. Der dritte Aufruf vergleicht die ersten fünf Komponenten von *Vec1* mit den Komponenten 6..10 von *Vec2*, der vierte vergleicht schließlich *Vec1[1]* mit dem Feld *P.x* und *Vec1[2]* mit dem Feld *P.y*.

Prozedurtypen

Als Erweiterung des Sprachstandards ermöglicht Turbo Pascal die Behandlung von Prozeduren und Funktionen als *Objekte* des Programms (und nicht als unveränderlicher Teil von ihm). Eine neue Klasse von Variablen – die *Prozedur-Variablen* – speichert nicht nur (wie Zeiger) die Adresse einer Routine, sondern hat für den Compiler auch die restlichen Charakteristika einer Prozedur bzw. Funktion. Eine detaillierte Erklärung von Prozedurtypen finden Sie in Kapitel 3 »Typen«.

Prozedur-Variablen

Nach der Vereinbarung eines Prozedurtyps lassen sich entsprechende Variablen deklarieren – wenn wir von den Definitionen des letzten Abschnitts ausgehen, beispielsweise:

```
var
  SwapProcPtr: SwapProc;
  F: MathFunc;
  X, Y: Integer;   { zwei "normale" Variablen }
```

Technisch gesehen, unterscheiden sich derartige Variablentypen in keinem Punkt von normalen Zeigern: sie enthalten eine Adresse und belegen vier Bytes im Speicher.

Die Fortsetzung unseres Beispiels:

```
procedure Swap(var A, B: Integer); far;
var
  Temp: Integer;
begin
  Temp := A; A := B; B:= Temp;
end;

function Tan(Winkel: Real): Real; far;
begin
  Tan:= Sin(Winkel) / Cos(Winkel);
end;

begin
  SwapProcPtr := Swap;  { Zuweisung an SwapProcPtr }
  SwapProcPtr(X,Y);     { Aufruf der Prozedur Swap }
  Swap(X,Y);            { dito }
  F := Tan;             { Zuweisung an Tan }
  WriteLn(F(X));        { Aufruf der Funktion Tan }
end.
```

Abgesehen davon, daß der Aufruf über *SwapProcPtr* mit einer normalen Pascal-Anweisung erfolgt, erschöpfen sich die Unterschiede zu Zeigern des Typs *Pointer* in zwei Punkten:

- in der Art der Zuweisung, die *direkt* (also ohne den Adreß-Operator @) erfolgt; und in der Typisierung: eine Prozedur-Variable enthält für den Compiler nicht »irgendeine« Adresse, sondern dieselben Charakteristika wie die Deklaration einer Prozedur bzw. Funktion.

Aufgrund der zweiten Eigenschaft lassen sich hier dieselben Prüfmechanismen wie bei »normalen« Aufrufen anwenden – das Ergebnis einer Anweisung wie

```
SwapProcPtr(X);
```

ist die Meldung des Compilers, daß hier ein Parameter zuwenig angegeben wurde.

Dieselbe Prüfung wird bei Zuweisungen an Prozedur-Variablen vorgenommen: »Zuweisungskompatibilität« bedeutet hier, daß Routine und Variable dieselbe Anzahl von Parametern mit demselben Typ und in derselben Reihenfolge sowie (bei Funktionen) denselben Ergebnistyp deklarieren müssen.

Technische Voraussetzungen

Damit sich eine Routine einer Prozedur-Variablen zuweisen läßt, **muß** sie als **far** deklariert sein (was sich auch über den Compiler-Befehl **{$F+}** im Quelltext erreichen läßt). Die folgenden Arten von Routinen lassen sich *nicht* an Prozedur-Variablen zuweisen bzw. über Prozedur-Variablen aufrufen:

- eingeschachtelte Prozeduren und Funktionen (weil sie nicht global verfügbar sind);
- **inline**-Routinen (weil sie vom Compiler als »Makro« behandelt werden und deshalb keine Adresse haben);
- **interrupt**-Routinen (weil sie mit einem speziellen RET-Befehl enden, für den ein weiterer Prozedurtyp notwendig wäre);
- die Prozeduren und Funktionen der Unit *System*, d.h. Standardfunktionen wie *Writeln*, *Read* (aufgrund ihrer variablen Parameterzahl), *Ord*, *Succ* usw. (weil sie vom Compiler direkt berechnet bzw. als direkte Befehle eingefügt werden).

Wenn Sie Standardfunktionen dennoch über Prozedur-Variablen aufrufen wollen, müssen Sie eine eigene Routine schreiben, die den Systemaufruf enthält und diese Routine einer Prozedur-Variablen zuordnen:

```
type
  IntProc = procedure(X: Integer);

var
  WriteProc: IntProc;

procedure WriteInt(Number: Integer); far;
begin
  Write(Number);
end;

begin
  WriteProc := WriteInt;   { Zuordnung }
  WriteProc(10);    {Aufruf von WriteInt über die Prozedur-Variable}
end.
```

Hinweis: Lokal definierte Routinen sind grundsätzlich als **near** codiert und lassen sich deshalb nicht über Prozedur-Variablen aufrufen. Der Versuch, nach einer Deklaration wie

```
procedure Aussen;

  procedure Innen;
  begin
    Writeln('"Innen" ist in "Außen" eingeschachtelt!');

  end; { Innen }
begin  { Aussen }
  Innen;
end;
```

die Prozedur *Innen* einer Prozedur-Variablen zuzuordnen, scheitert am Widerstand des Compilers (der damit den Eigenheiten der Prozessorfamilie 80x86 bei *near*-Adressierungen Rechnung trägt).

Prozedur-Variablen in Strukturen

Da eine Prozedur-Variable (abgesehen von ihrer »Charakteristik« für den Compiler) letztendlich ein ganz normales Datum darstellt, läßt sie sich wie jede andere Variable auch als Element eines Arrays oder als Feld eines Records verwenden:

```
type
  GotoProc  = procedure(X, Y: Integer);
  ProcList  = array[1..10] of GotoProc;

  WindowPtr = ^Window;
  Window    = record
                Next: WindowPtr;
                Title: string[20];
                Top, Left, Bottom, Right: Integer;
                SetCursor: GotoProc;
              end;
var
  P : ProcList;
  W : WindowPtr;
```

Nach diesen Deklarationen sind die folgenden Prozeduraufrufe möglich:

```
P[3](1,1);              { ruft die Prozedur des Typs GotoProc auf,
die
                          als drittes Element im Array P gespeichert
                          ist, und übergibt die Werte (1,1) }
W^.SetCursor(10,10);    { ruft die Prozedur des Typs GotoProc auf,
                          die als Feld SetCursor eines Records des
                          Typs Window gespeichert ist, auf den W
                          zeigt, und übergibt die Werte (10,10) }
```

Prozedur-Variablen als Parameter

Die Behauptung, daß Prozedur-Variablen eigentlich völlig normale Daten darstellen, gilt nicht nur für die Aufnahme in Strukturen. Das folgende Programm demonstriert, wie sich einer Prozedur drei verschiedene Funktionen als *Parameter* übergeben lassen:

```
program Tables;

type
  Func = function(X,Y: Integer): Integer;

function Add(X,Y: Integer): Integer; far;
begin
  Add := X + Y;
end;

function Mult(A,B: Integer): Integer; far;
begin
  Mult := A * B;
end;

function Funny(X,Y: Integer): Integer; far;
begin
  Funny := (X+Y) * (X-Y);
end;
```

Diese drei Funktionen sollen über Prozedur-Variablen aufgerufen werden und müssen deshalb als **far** codiert sein; der Rest des Programms verwendet dagegen **near**-Aufrufe.

```
procedure PrintTable(W, H: Integer; Operation: Func);
var  x,y: Integer;
begin
  for y:= 1 to H do
  begin
    for x := 1 to W do
      Write(Operation(X,Y):5);
    Writeln;
  end;
end;
```

PrintTable erwartet neben der Zeilen- und Spaltenzahl der Tabelle eine Prozedur-Variable des Typs *Func* als Parameter. Wie der Aufruf von *Write* zeigt, lassen sich Prozedur-Variablen auch bei Funktionen direkt anstelle eines Funktionsbezeichners verwenden: der Aufruf über *Operation* aktiviert die entsprechende Funktion und liefert deren Ergebnis zurück.

```
begin
  PrintTable(10,10,Add);
  PrintTable(10,10,Mult);
  PrintTable(10,10,Funny);
end.
```

Das Hauptprogramm ruft *PrintTable* dreimal auf, wobei jeweils eine andere Funktion als *Operation* übergeben wird. Der zweite Aufruf produziert beispielsweise das folgende Ergebnis:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

Prozedur-Variablen als Parameter können Programme speziell dann wesentlich vereinfachen, wenn zwischen einer größeren Zahl ähnlicher Aktionen unterschieden werden muß – wie eben den Grundrechenarten oder der Bewegung des Cursors in einem Textverarbeitungsprogramm (was sonst über eine schier endlose Folge von **case**-Zweigen gelöst werden muß).

Als Parameter übergebene Prozedur-Variablen müssen den im Abschnitt »Technische Voraussetzungen« aufgelisteten Regeln genügen: Die übergebene Routine muß als **far** codiert und in ihrer Deklaration zuweisungskompatibel zum entsprechenden formalen Parameter sein; die Routinen von *System*, **inline**- und **interrupt**-Prozeduren sowie lokale Funktionen und Prozeduren lassen sich nicht als Parameter verwenden.

9. Programme und Units

Die Syntax eines Programms

Ein Turbo Pascal-Programm hat eine ähnliche Form wie eine Prozedurdeklaration. Die Unterschiede liegen lediglich im Programmkopf und der Möglichkeit einer zusätzlichen **uses**-Anweisung.

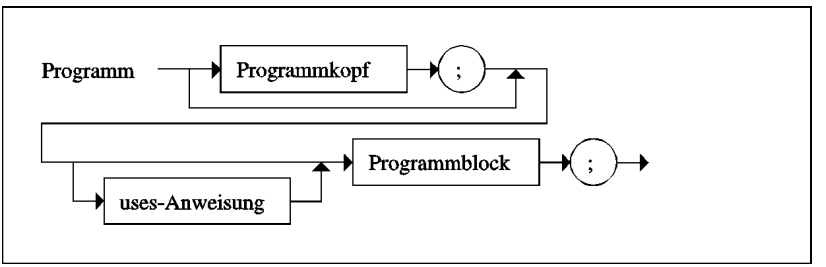


Abbildung 9.1 Syntax eines Turbo Pascal-Programms

Der Programmkopf

Der Programmkopf legt Namen und Parameter eines Programms fest.

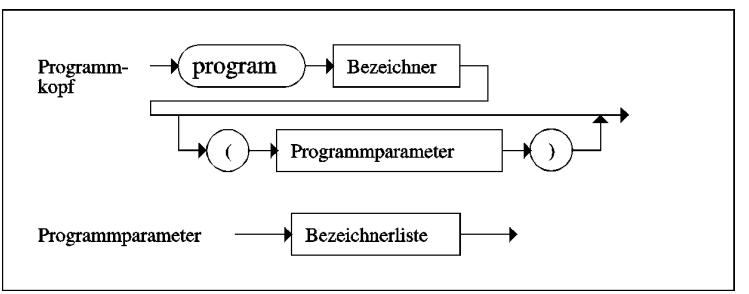


Abbildung 9.2 Syntax eines Programmkopfes

Programmparameter werden in Turbo Pascal vom Compiler ignoriert. Die folgenden Programmköpfe sind also identisch:

```
program Test(Input, Output);
program Test;
```

Die Anweisung uses

legt fest, welche Units ein Programm benutzt. Hier müssen nur direkt benutzte Units aufgeführt werden: Compiler und Linker prüfen bei der Aufnahme einer Unit automatisch, ob diese Unit ihrerseits andere Units voraussetzt, und binden sie gegebenenfalls in das Gesamtprogramm ein.

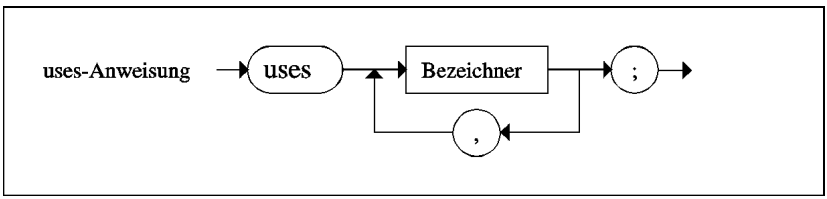


Abbildung 9.3 Syntax einer uses*-Anweisung*

Die Unit *System* wird automatisch vom Compiler in Programme eingefügt. Sie implementiert alle Laufzeitfunktionen des Systems auf niedriger Ebene, u.a. die Behandlung von Strings und Mengen, die dynamische Speicherbelegung und die Bereichsüberprüfung.

Darüber hinaus verfügt Turbo Pascal über eine Reihe von Standard-Units wie *WinDos*, *WinTypes* und *WinProcs*. Ihre Aufnahme erfolgt nicht automatisch – sie müssen jeweils explizit über eine **uses**-Anweisung angegeben werden:

```
uses WinTypes, WinProcs;  { macht die Definitionen von WinTypes und
WinProcs verfügbar }
```

Die Reihenfolge der Unit-Namen in der **uses**-Anweisung bestimmt deren Initialisierungs-Reihenfolge (vgl. auch »Der Initialisierungs-Teil« in diesem Kapitel).

Die Syntax von Units

Units sind die Grundlage der modularen Programmierung in Turbo Pascal. Sie werden zur Erzeugung von Bibliotheken benutzt, die in verschiedene Programme eingebunden werden können, ohne den Quelltext selber zugänglich zu machen. Mit ihrer Hilfe läßt sich ein Programm in Module unterteilen, die nach der logischen Anordnung ihrer Bestandteile zusammengestellt werden.

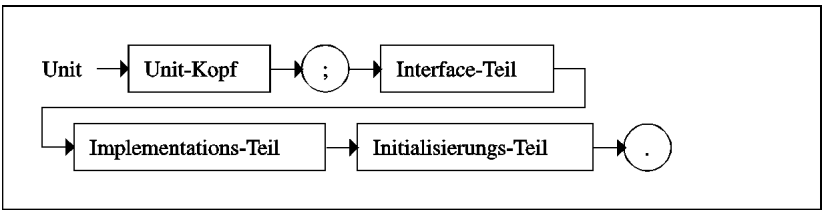


Abbildung 9.4 Syntax einer Unit

Der Kopf einer Unit

legt den Namen der Unit fest, der innerhalb des gesamten Programms »einzigartig« sein muß – mehrere Units mit demselben Namen sind innerhalb eines Programms nicht möglich.

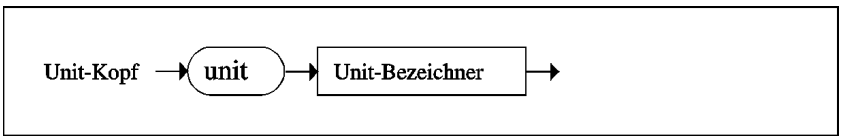


Abbildung 9.5 Syntax eines Unit-Kopfes

Der Interface-Teil

einer Unit deklariert alle »öffentlichen« Typen, Variablen, Prozeduren und Funktionen. Die dazugehörigen Bezeichner sind für das Programm und jede Unit zugänglich, das diese Unit benutzt. Programme und Units können darauf in derselben Weise zugreifen, als wären sie in einem übergeordneten Block deklariert worden.

Solange eine Prozedur oder Funktion nicht als **inline** deklariert ist, enthält der Interface-Teil nur ihren Kopf. Der Block der Prozedur oder Funktion befindet sich im Implementations-Teil der Unit.

Der Implementations-Teil

definiert die Anweisungsblöcke (»Rümpfe«) aller »öffentlichen« Prozeduren und Funktionen, solange diese nicht als **inline** deklariert sind. Zusätzlich werden hier alle »privaten« Konstanten, Variablen, Prozeduren und Funktionen deklariert, die nur innerhalb der Unit, nicht aber von anderen Programmen oder Units benutzt werden können.

Die Auflistung der Funktions- und Prozedurköpfe im Interface-Teil entspricht im Prinzip einer **forward**-Deklaration, nur daß der Zusatz **forward** hier nicht

explizit aufgeführt wird. Auf im Interface-Teil deklarierte Funktionen und Prozeduren kann deshalb im Implementations-Teil ohne Rücksicht auf ihre Reihenfolge im Quelltext Bezug genommen werden.

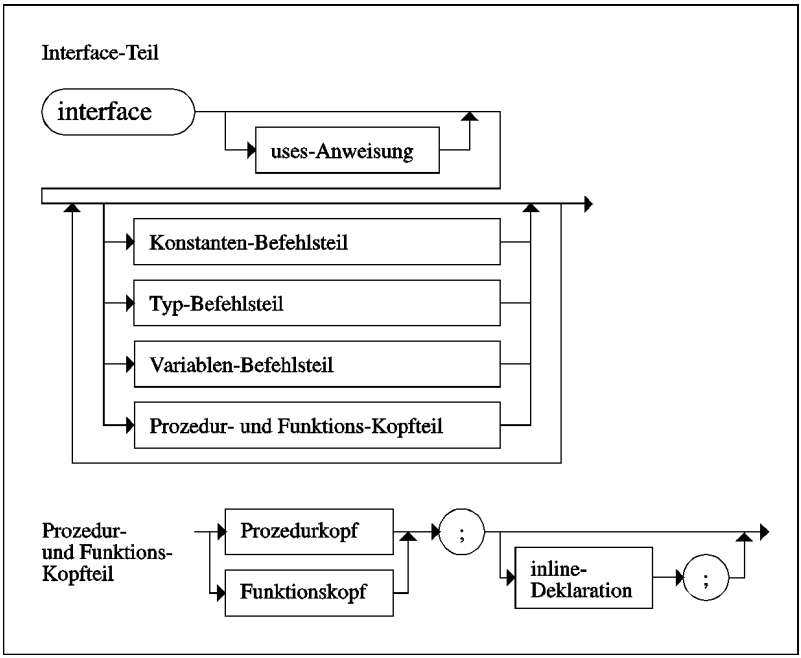


Abbildung 9.6 Syntax des Interface-Teils einer Unit

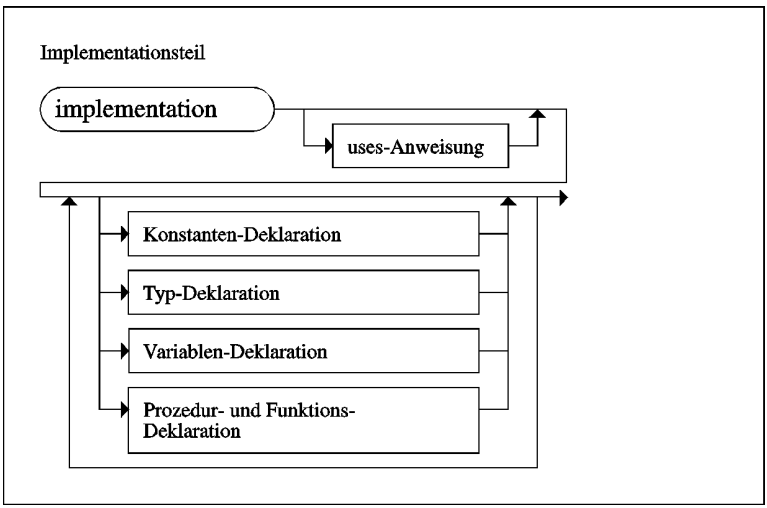


Abbildung 9.7 Syntax des Implementations-Teils einer Unit

Die Funktions- und Prozedurblöcke, deren Köpfe im Interface-Teil deklariert sind, werden im Implementations-Teil nur durch die Angabe ihres Bezeichners eingeleitet.

Hinweis: Im Gegensatz zu **forward** erlaubt Turbo Pascal hier eine Wiederholung der formalen Parameter. Turbo Pascal nimmt dann eine Prüfung auf Übereinstimmung vor (und meldet einen Fehler, wenn angegebene Parameterlisten zwischen **Interface** und **Implementation** differieren).

Der Initialisierungs-Teil

ist der letzte Bestandteil einer Unit und stellt das »Hauptprogramm« dar. Er besteht entweder aus dem reservierten Wort **end** (nämlich dann, wenn kein ausführbares »Hauptprogramm« vorhanden ist), oder aus einem mit begin eingeleiteten Block, der Anweisungen zur Initialisierung enthalten kann und durch das abschließende end der Unit beendet wird.

Die »Hauptprogramme« der Units, die ein Programm benutzt, werden beim Start des Programms in der Reihenfolge ausgeführt, in der die Unitnamen in der **uses**-Anweisung dieses Programms erscheinen.

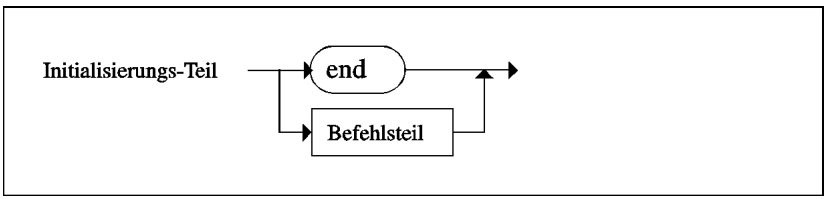


Abbildung 9.8 Syntax des Initialisierungs-Teils einer Unit

Units, die andere Units voraussetzen

Eine **uses**-Anweisung am Anfang einer Unit kann andere Units angeben, die von dieser Unit benutzt bzw. vorausgesetzt werden. Je nach Art der Verkettung ist diese Benutzung direkt oder indirekt:

```
unit Unit1;          unit Unit2;          program Hauptprogramm;
interface            interface            uses Unit2;
const c = 1;         uses Unit1;          const a = b;
implementation       const b = c;         begin
const d = 2;         implementation       end.
end.                 end.
```

Hier wird *Unit1* von *Unit2* direkt vorausgesetzt und muß deshalb in *Unit2* mit **uses** aufgeführt werden; analoges gilt für *Hauptprogramm* und *Unit2*. Die Beziehung zwischen *Unit1* und *Hauptprogramm* ist dagegen indirekt – sie wird vom Linker automatisch berücksichtigt.

Nach der Veränderung des Interface-Teils einer Unit müssen alle Units neu compiliert werden, die diese Unit direkt benutzen. Wenn in unserem Beispiel der Interface-Teil von *Unit1* verändert wird (z.B. in *c = 2*), dann ist eine erneute Compilierung von *Unit2* erforderlich.

Veränderungen von Units, die sich rein auf den Implementations- und/oder Initialisierungsteil beschränken, können dagegen isoliert durchgeführt werden. Im gegebenen Beispiel könnte der Implementations-Teil von *Unit1* verändert werden (z.B. in *d = 1*), ohne daß danach *Unit2* neu compiliert werden müßte.

Bei der Compilierung einer Unit berechnet Turbo Pascal eine *Unit-Versionsnummer*, die sich als Prüfsumme des Interface-Teils ergibt. Bezogen auf unser Beispiel bedeutet das: bei der Compilierung von *Unit2* wird die aktuelle Versionsnummer von *Unit1* in der compilierten Version von *Unit2* gespeichert. Bei der Compilierung von *Hauptprogramm* vergleicht Turbo Pascal die in *Unit2* gespeicherte Nummer mit dem tatsächlichen Inhalt von *Unit1*. Stimmen die Nummern nicht überein, dann gibt der Compiler (je nach Art der Compilierung) eine Fehlermeldung aus bzw. compiliert *Unit2* neu.

uses-Anweisungen im Implementationsteil und gegenseitige Abhängigkeiten

Eine **uses**-Anweisung im Interface-Teil wie

```
uses MyUnit;
```

macht die globalen Bezeichner der Unit *MyUnit* innerhalb der gesamten Unit verfügbar und legt damit gleichzeitig eine strenge Hierarchie fest: Wenn *UnitA* eine weitere Unit *UnitB* mit **uses** aufführt, gibt es keine Möglichkeit, *UnitB* ihrerseits mit **uses** in *UnitA* aufzunehmen.

Aus diesem Grund erlaubt Turbo Pascal zusätzliche **uses**-Anweisungen im Implementationsteil: In dieser Weise aufgenommene Units sind sozusagen »lokal« und kommen sich auch dann nicht in die Quere, wenn sie sich gegenseitig voraussetzen. Das folgende Beispiel demonstriert eine solche Situation, die sich ansonsten nur mit zwei Versionen ein und derselben Routine lösen ließe:

```
program Im_Kreis;

uses  Crt, Display;  { Die Unit Display enthält die Routine WriteXY }

const  Msg = 'Zurück in bewohnte Gebiete';

begin
  ClrScr;
  WriteXY(1,1, 'Obere linke Ecke des Bildschirms');
  WriteXY(100,100, 'Ziemlich unwahrscheinlich...');
  WriteXY(81 - Length(Msg), 15, Msg);
end.
```

Im_Kreis verwendet eine Routine namens *WriteXY*, die den Text ab der

angegebenen Position auf den Bildschirm ausgibt und in der Unit *Display* definiert ist:

```
unit Display;

interface
procedure WriteXY(X,Y: Integer; Msg: string);

implementation
uses
  Crt, Error;

procedure WriteXY(X,Y: Integer; Msg: string);
begin
  if (X > 0) and (X <= 81-Length(Msg)) and (Y > 0) and (Y <= 25) then
  begin
    GotoXY(X,Y);
    Write(Msg);
  end
   else PrintError('X/Y-Koordinaten außerhalb des Bildschirms!');
end;

end.
```

Die Routine *WriteXY* der Unit *Display* prüft, ob der als *Msg* übergebene String komplett auf den Bildschirm paßt – falls ja, wird er ausgegeben; falls nein, ruft sie eine Routine namens *PrintError* zur Ausgabe einer Fehlermeldung aus.

PrintError ist in einer Unit namens *Error* definiert, die von *Display* folgerichtig auch mit **uses** aufgeführt wurde:

```
Unit Error;

interface

procedure PrintError(ErrMsg: string);

implementation

uses
  Display;

procedure PrintError(ErrMsg: string);
begin
  WriteXY(1,25, 'Fehler: ' + ErrMsg);
end;

end.
```

Und hier würden die Probleme mit »öffentlichen« **uses**-Anweisungen anfangen: *PrintError* verwendet seinerseits die Routine *WriteXY* zur Ausgabe der Fehlermeldung.

Durch das »Verschieben« beider **uses**-Anweisungen in den Implementationsteil kann Turbo Pascal die Interface-Teile komplett compilieren und damit den Aufruf von *WriteXY* in den Code von *PrintError* einsetzen, weil der Compiler zu diesem Zeitpunkt bereits »weiß«, wie die Deklaration von *WriteXY* aussieht.

Dieser Mechanismus gilt auch für mehrfache gegenseitige Abhängigkeiten (d.h. für Fälle, in denen sich drei oder mehr Units »kreisförmig« aufeinander beziehen). Als Definition formuliert:

Solange die Interface-Teile zweier Units nicht gegenseitig voneinander abhängig sind (z.B. aufgrund gemeinsamer Daten-Deklarationen), können sich Units gegenseitig voraussetzen: die entsprechenden **uses**-Anweisungen werden im Implementationsteil untergebracht; sie müssen dort unmittelbar auf das reservierte Wort **implementation** folgen.

Gemeinsame Datendeklarationen

Die Definition des vorangegangenen Abschnitts läßt gemeinsam benutzte Daten (und Routinen) bewußt aus. Tatsächlich müßten von beiden Units verwendete Variablen, Datentypen oder Routinen im Interface-Teil eines der beiden stehen – womit wir wieder beim alten Problem angelangt wären. Setzen wir beispielsweise einmal voraus, daß die Prozeduren *WriteXY* und *PrintError* unseres Programms *Im_Kreis* zusätzlich die Grenzen eines Ausgabefensters als Parameter erwarten, also:

```
procedure WriteXY(OutputWindow: WindRec;  { Fenstergrenzen }
                  X,Y: Integer;           { Position im Fenster }
                  Msg: string;)           { Text }

procedure PrintError(OutputWindow: WindRec;  { Fenstergrenzen }
                     X,Y: Integer;           { Position im Fenster }
                     Msg: string;)           { Text }
```

Da der Datentyp *WindRec* für beide Routinen zur Verfügung stehen muß, gibt es hier nur eine (aber recht einfache) Lösung: Definieren Sie eine dritte Unit, die nur die entsprechende Typ-Definition enthält:

```
unit WindData;
interface
type
  WindRec = record
              X1,Y1,X2,Y2: Integer;  { Fenstergrenzen }
              TColor, BkColor:  Integer;  { Textattribut }
              Active: Boolean;
            end;
implementation
end.
```

Weil diese Unit keine der beiden anderen voraussetzt, läßt sie sich ohne jedes Problem mit einer **uses**-Anweisung in die Interface-Teile von *Display* und *Error* aufnehmen. (Sowohl *Display* als auch *Error* arbeiten dann mit zwei **uses**-Anweisungen – einer im Interface- und einer im Implementationsteil).

10. Dynamische Linkbibliotheken

In der Windows-Umgebung ermöglichen dynamische Linkbibliotheken (DLLs), daß mehrere Anwendungsprogramme denselben Programmcode und dieselben Ressourcen nutzen. Mit Turbo Pascal für Windows können Sie existierende DLLs verwenden oder Ihre eigenen DLLs schreiben, die dann auch von anderen Programmen verwendet werden können.

Was ist eine DLL?

Eine DLL ist ein ausführbares Modul, das sowohl Programmcode als auch Ressourcen enthält, die von anderen Anwendungen oder DLLs verwendet werden können. Im traditionellen Turbo-Pascal-Konzept entsprechen diese DLLs am ehesten den Units: beide Module stellen einem Programm »Dienstleistungen« in Form von Prozeduren und Funktionen zur Verfügung. Dieser Vergleich soll aber lediglich als Brücke dienen, denn tatsächlich gibt es einige Unterschiede zwischen DLLs und Units. Einer der wichtigsten ist, daß Units *statisch eingebunden* werden, DLLs hingegen *dynamisch*.

Wenn ein Programm eine Prozedur oder Funktion aus einer Unit verwendet, wird eine Kopie des Codes dieser Prozedur oder der Funktion statisch in die ausführbare Datei des Programms eingebunden. Würden zwei Programme gleichzeitig ausgeführt werden, die dieselbe Prozedur oder Funktion einer Unit verwenden, würden auch zwei Kopien dieser Routine im System angelegt. Es wäre jedoch wesentlich effizienter, wenn die beiden Programme auf nur eine Kopie dieser Routine zugreifen könnten. DLLs ermöglichen genau dies.

Im Gegensatz zu Units wird der Programmcode einer DLL nicht in das Programm eingebunden, das die DLL verwendet. Vielmehr befindet sich dieser Code zusammen mit den eventuell notwendigen Ressourcen in einer separaten Datei mit der Dateinamenserweiterung .DLL. Diese Datei muß verfügbar sein, wenn das Client-Programm ausgeführt wird. Ruft dieses Programm nun eine in der DLL definierte Prozedur oder Funktion auf, verbindet die Programmladefunktion von Windows diese Aufrufe mit den entsprechenden Einsprungspunkten in der DLL.

Eine DLL muß nicht in Turbo Pascal geschrieben sein, um von einer Turbo-Pascal-Anwendung verwendet werden zu können. Außerdem können auch

Programme, die in anderen Programmiersprachen geschrieben wurden, DLLs verwenden, die in Turbo Pascal geschrieben wurden. DLLs eignen sich daher besonders für Programmprojekte, bei denen in mehreren Programmiersprachen gearbeitet wird.

Verwendung von DLLs

Damit ein Modul eine Prozedur oder Funktion aus einer DLL verwenden kann, muß das Modul die Prozedur oder Funktion mit Hilfe der **external**-Deklaration importieren. Die folgende Deklaration importiert zum Beispiel eine Funktion namens *GlobalAlloc* aus einer DLL namens »KERNEL« (dem Windows-Kernel).

```
function GlobalAlloc (Flags:Word; Bytes:Longint) : THandle; far;
external 'KERNEL' index 15;
```

In importierten Prozeduren und Funktionen nimmt die **external**-Anweisung die Stelle der sonst präsenten Deklarations- und Anweisungs-Abschnitte ein. Importierte Prozeduren und Funktionen müssen **far**-Aufrufe verwenden, indem Sie die Routine als **far** deklarieren oder den {**$F+**}-Compiler-Befehl verwenden. Ansonsten verhalten sie sich wie normale Prozeduren und Funktionen.

In Turbo Pascal für Windows gibt es drei Möglichkeiten, um Prozeduren und Funktionen zu importieren:

- über den Namen
- über einen neuen Namen
- über einen Ordinalwert

Das Format der **external**-Deklaration für alle drei Methoden wird im folgenden Beispiel beschrieben.

Ist keine **index**- oder name-Anweisung angegeben, wird die Prozedur über den Namen importiert. Der verwendete Name ist derselbe wie der Bezeichner der Prozedur oder Funktion. Im folgenden Beispiel wird die Prozedur *ImportByName* aus *TESTLIB* über den Namen *IMPORTBYNAME* importiert:

```
procedure ImportByName; external 'TESTLIB';
```

Ist eine **name**-Anweisung angegeben, wird die Prozedur oder Funktion mit einem anderen Namen als dem Bezeichner importiert. Im folgenden Beispiel wird die Prozedur *ImportByName* aus *TESTLIB* über den Namen *REALNAME* importiert:

```
procedure ImportByName; external 'TESTLIB' name 'REALNAME';
```

Wenn eine **index**-Anweisung vorliegt, wird die Prozedur oder Funktion über einen Ordinalwert importiert. Der Import über Ordinalwerte reduziert die benötigte Ladezeit des Moduls, da Windows den Namen nicht in der Namen-

stabelle der DLL suchen muß. Im folgenden Beispiel wird die Prozedur *ImportByOrdinal* als fünfter Einsprungspunkt einer DLL namens *TESTLIB* importiert:

```
procedure ImportByOrdinal; external 'TESTLIB' index 5;
```

Der DLL-Name, der nach dem Schlüsselwort **external** angegeben wird, und der neue Name in der **name**-Anweisung müssen keine String-Literale sein. Alle konstanten Stringausdrücke sind zulässig. Genauso kann die Ordinalzahl, die in der **index**-Anweisung angegeben wird, ein beliebiger konstanter Integerausdruck sein.

```
const
  TestLib = 'TESTLIB';
  Ordinal = 5;

procedure ImportByName;    external TestLib;
procedure ImportByNewName; external TestLib name 'REALNAME';
procedure ImportByOrdinal; external TestLib index Ordinal;
```

Obwohl DLLs Variablen enthalten können, ist es nicht möglich, sie in andere Module zu importieren. Jeder Zugriff auf die Variablen einer DLL muß über eine prozedurale Schnittstelle stattfinden.

In Units importieren

Deklarationen importierter Prozeduren und Funktionen können direkt in dem Programm untergebracht werden, das sie importiert. Normalerweise werden sie aber in einer »Import-Unit« zusammengefaßt, die die Deklarationen für alle Prozeduren und Funktionen in einer DLL enthält. Dazu gehören auch Konstanten und Typen, die zum Zugriff auf die DLL notwendig sind. Die in Turbo Pascal für Windows enthaltenen Units *WinTypes* und *WinProcs* sind Beispiele für solche Import-Units. Sie sind für den Zugriff auf eine DLL nicht absolut notwendig, erleichtern aber die Verwaltung von Projekten, die mehrere DLLs verwenden.

Nehmen wir als Beispiel eine DLL namens DATETIME.DLL, die vier Routinen definiert, um Datum und Zeit abzurufen und einzustellen. Diese DLL verwendet dabei einen Recordtyp, der Tag, Monat und Jahr, und einen anderen, der Sekunden, Minuten und Stunden speichert. Anstatt nun die jeweiligen Prozedur-, Funktions- und Typendeklarationen in jedem Programm anzugeben, das die DLL verwendet, können Sie eine Import-Unit erstellen, die zusammen mit der DLL verwendet wird.

```
unit DateTime;

interface

type
  TTime Rec = record
    Second: Integer;
    Minute: Integer;
```

```
    Hour: Integer;
  end;

type
  TDateRec = record
    Day: Integer;
    Month: Integer;
    Year: Integer;
  end;

procedure SetTime (var Time: TTimeRec);
procedure GetTime (var Time: TTimeRec);
procedure SetDate (var Date: TDateRec);
procedure GetDate (var Date: TDateRec);

implementation

procedure SetTime; external 'DATETIME' index 1;
procedure GetTime; external 'DATETIME' index 2;
procedure SetTime; external 'DATETIME' index 3;
procedure GetTime; external 'DATETIME' index 4;

end.
```

Bei jedem Programm, das Routinen aus DATETIME.DLL verwendet, kann dann einfach *DateTime* in einer **uses**-Anweisung angegeben werden.

```
program ShowTime;

uses WinCrt, DateTime;

var
  Time: TTimeRec;

begin
  GetTime (Time);
  with Time do
    WriteLn ('Die Uhrzeit ist ', Hour, ':', Minute,':', Second);
end.
```

Ein weiterer Vorteil von Import-Units wie *DateTime* ist, daß bei Modifikation der zugehörigen DATETIME.DLL nur eine Unit, die Import-Unit *DateTime*, dieser Modifikation entsprechend angepaßt werden muß.

Statischer und dynamischer Import

Die **external**-Deklaration ermöglicht den statischen Import von Prozeduren und Funktionen aus einer DLL. Eine statisch importierte Prozedur oder Funktion bezieht sich immer auf denselben Einsprungspunkt in derselben DLL. Windows unterstützt ebenfalls dynamischen Import. Hierbei wird der Name der DLL und der Name bzw. die Ordinalzahl der importierten Prozedur und Funktion wäh-

rend der Laufzeit des Programms angegeben. Das Programm *ShowTime* verwendet einen dynamischen Import, um die Prozedur *GetTime* in der DLL DATETIME.DLL aufzurufen. Beachten Sie, daß eine prozedurale Typvariable verwendet wird, um die Adresse der Prozedur *GetTime* anzugeben.

```
program ShowTime;

uses WinProcs, WinTypes, WinCrt;

type
  TTimeRec = record
    Second: Integer
    Minute: Integer
    Hour: Integer
  end;
  TGetTime = procedure (var Time: TTimeRec);

var
  Time: TTimeRec;
  Handle: THandle;
  GetTime: TGetTime;

begin
  Handle:= LoadLibrary ('DATETIME.DLL');
  if Handle = 32 then
  begin
    GetTime :=TGetTime (GetProcAddress (Handle, 'GETIME'));
    if @GetTime  nil then
    begin
      GetTime (Time);
      with Time do
        WriteLn ('Die Uhrzeit ist ', Hour, ':', Minute,':', Second);
    end;
    FreeLibrary (Handle);
  end;
end;
```

Erstellen von DLLs

Die Struktur einer Turbo-Pascal-DLL entspricht der eines Programms, ausgenommen die Tatsache, daß eine DLL mit einer **library**-Kopfzeile beginnt anstatt mit einer **program**-Kopfzeile. Das reservierte Wort **library** gibt Turbo Pascal an, eine ausführbare Datei zu erstellen, die die Dateinamenserweiterung .DLL anstatt .EXE hat. Außerdem stellt sie sicher, daß die ausführbare Datei als DLL gekennzeichnet wird.

Das folgende Beispiel implementiert eine sehr einfache DLL mit den zwei

Export-Funktionen *Min* und *Max*, die den größeren bzw. kleineren zweier ihnen übergebener Integer-Werte zurückgeben.

```
library MinMax;

function Min (X, Y: Integer): Integer; export;
begin
  if X < Y then Min := X else Min := Y;
end;

function Max (X, Y: Integer): Integer; export;
begin
  if X > Y then Max := X else Max := Y;
end;

exports
  Min index 1;
  Max index 2;

begin
end.
```

Beachten Sie in diesem Beispiel die Verwendung der **export**-Deklaration, um *Min* und *Max* für den Export vorzubereiten, und die Anweisung **exports**, um die beiden Routinen tatsächlich zu exportieren, wobei für beide Routinen Ordinalzahlen bereitgestellt werden.

Obwohl das oben dargestellte Beispiel dies nicht zeigt, können Bibliotheken (DLLs) mehrere Units enthalten. In solchen Fällen wird die Bibliotheks-Quelldatei meist auf eine uses-Anweisung, eine **exports**-Anweisung und den Initialisierungscode der Bibliothek reduziert. Im folgenden finden Sie dazu ein Beispiel:

```
library Editors;

uses EdInit, EdInOut, EdFormat, EdPrint;

exports
  InitEditors index 1,
  DoneEditors index 2,

InsertText index 3
DeleteSelection index 4,
FormatSelection index 5,
PrintSelection index 6,
.
.
SetErrorHandler index 53;

begin
 InitLibrary;
end.
```

Die Deklaration export

Wenn Prozeduren und Funktionen von einer DLL exportiert werden sollen, müssen sie mit dem Zusatz **export** compiliert werden. Die **export**-Deklaration gehört zur selben Gruppe wie die Deklarationen **near**, **far**, **inline** und **interrupt**. Dies bedeutet, daß eine **export**-Deklaration, falls vorhanden, bei der ersten Einführung der Prozedur oder Funktion angegeben werden muß. Sie kann nicht in der definierenden Deklaration einer **forward**-Routine angegeben werden.

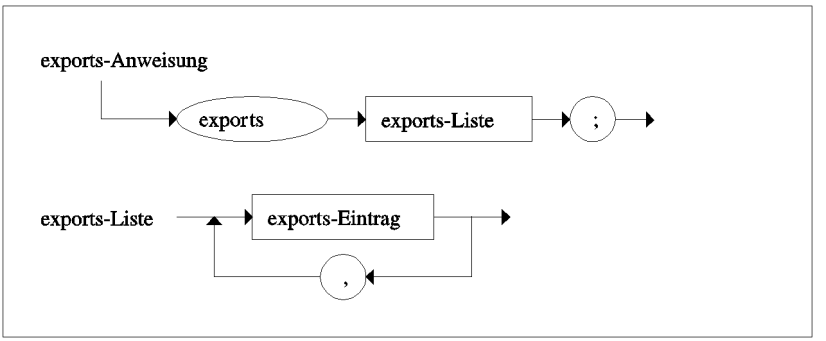


Abbildung 10.1 Syntax einer export-Deklaration

Die **export**-Deklaration macht eine Prozedur oder Funktion »exportierbar«. Solche Routinen werden als far aufgerufen, und der Compiler generiert für sie einen speziellen Einsprungs- und Austrittscode. Beachten Sie aber, daß der tatsächliche Export der Prozedur oder Funktion erst erfolgt, wenn die Routine in der Anweisung **exports** der Bibliothek aufgeführt wird.

Die Anweisung exports

Eine Prozedur oder Funktion wird von einer DLL exportiert, wenn sie in der **exports**-Anweisung der Bibliothek aufgeführt ist.

Eine **exports**-Anweisung kann überall und beliebig oft in einem Programm oder dem Deklarationsteil einer Bibliothek vorkommen. Jeder Eintrag in einer **exports**-Anweisung gibt die Bezeichner einer zu exportierenden Prozedur oder Funktion an. Der Bezeichner muß eine Prozedur oder Funktion kennzeichnen, die mit der Direktive **export** compiliert wurde. Der Bezeichner kann falls notwendig in seiner vollständigen Form angegeben werden, also indem der Name der Unit, ein Punkt und dann der eigentliche Bezeichner der Routine aufgeführt werden.

Jeder Eintrag in der **exports**-Liste kann eine index-Anweisung enthalten, die aus dem Wort **index** gefolgt von einer Integer-Wert im Bereich von 1 und 32767 besteht. Ist eine solche **index**-Anweisung angegeben, verwendet die zu expor-

tierende Prozedur oder Funktion diese Ordinalzahl. Wenn keine index-Anweisung in einer exports-Eingabe existiert, wird die Ordinalzahl automatisch zugewiesen.

Ein Eintrag kann auch eine **name**-Anweisung enthalten, die aus dem Wort **name** gefolgt von einer String-Konstanten besteht. Wenn eine **name**-Anweisung angegeben wird, verwendet die zu exportierende Prozedur oder Funktion den mit der String-Konstanten angegebenen Namen. Wenn keine **name**-Anweisung in einer exports-Eingabe existiert, wird die Prozedur oder Funktion über ihren Bezeichner exportiert der in Großbuchstaben umgewandelt wird.

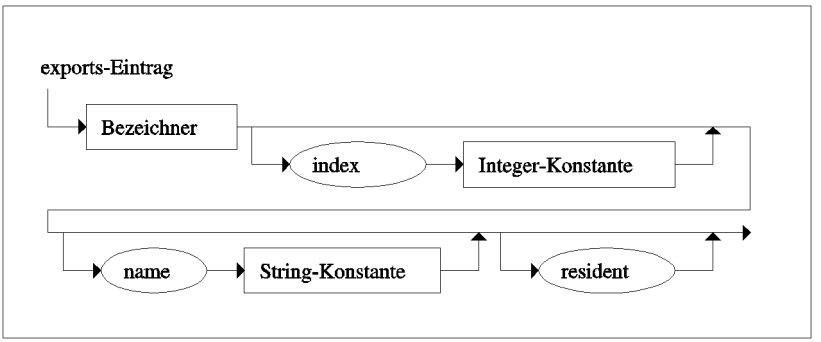


Abbildung 10.2 Syntax einer exports-Anweisung

Schließlich kann ein Eintrag in der **exports**-Liste das Schlüsselwort **resident** enthalten. Wenn**resident** angegeben wird, verbleiben die Exportinformationen im Speicher, während die DLL geladen wird. Die Option **resident** reduziert die Zeit wesentlich, die Windows benötigt, um nach einem DLL-Eintrag über dessen Namen zu suchen. Wenn also Client-Programme, die auf die DLL zugreifen, sehr wahrscheinlich bestimmte Einträge über Namen importieren werden, sollten die Einträge mit dem Schlüsselwort **resident** exportiert werden.

Ein Programm kann zwar eine **exports**-Anweisung enthalten, dies ist aber selten der Fall, da Windows es nicht zuläßt, daß Anwendungsprogramme Funktionen exportieren, die andere Anwendungen verwenden können.

Initialisierungs- und Austrittscode der Bibliothek

Der Anweisungsteil einer Bibliothek stellt den Bibliothek-Initialisierungscode dar. Der Initialisierungscode wird nur einmal ausgeführt, und zwar, wenn die Bibliothek das erste Mal geladen wird. Wenn die folgenden Anwendungen, die die Bibliothek verwenden, geladen werden, führt Windows den Initialisierungscode nicht noch einmal aus, sondern erhöht einfach die Benutzerzahl der DLL.

Eine DLL wird solange im Speicher gehalten, wie ihre Benutzerzahl größer als Null ist. Wenn sie Null wird, womit angegeben wird, daß alle Anwendungen, die die DLL verwendet haben, beendet sind, wird die DLL aus dem Speicher entfernt. An diesem Punkt werden dann die Austrittsprozeduren der Bibliothek

ausgeführt. Die Austrittsprozeduren werden mit der Variable *ExitProc* registriert. Weitere Informationen hierzu finden Sie in Kapitel 18 »Interne Details«.

Der Initialisierungscode einer DLL erledigt normalerweise Aufgaben, wie die Registrierung von Fensterklassen für Fensterprozeduren, die in der DLL enthalten sind, und die Einstellung der Anfangswerte für die globalen Variablen der DLL. Der Initialisierungscode einer Bibliothek kann eine Fehlerbedingung dadurch anzeigen, daß die Variable *ExitCode* auf Null gesetzt wird (*ExitCode* wird von der Unit *System* deklariert). Die Voreinstellung für *ExitCode* ist 1, was anzeigt, daß die Initialisierung erfolgreich war. Wenn der Initialisierungscode den *ExitCode* auf Null setzt, wird die DLL aus dem Systemspeicher entfernt.

Wenn die Austrittsprozeduren einer Bibliothek ausgeführt werden, enthält die Variable *ExitCode* keinen Code zur Beendigung, wie das bei Programmen der Fall ist. Anstatt dessen enthält *ExitCode* einen der Werte *wep_System_Exit* oder *wep_Free_DLL*. Diese sind in der Unit *WinTypes* definiert. *wep_System_Exit* gibt an, daß Windows beendet wird. *wep_Free_DLL* gibt an, daß nur diese einzelne DLL aus dem Speicher entfernt wird.

Im folgenden finden Sie ein Beispiel einer Bibliothek mit Initialisierungs- und Austrittsprozedur:

```
library Test;

uses WinTypes, WinProcs;

var
  SaveExit: Pointer;

procedure LibExit; far;
begin
  if ExitCode = wep_System_Exit then
  begin
    .
    .
  { Windows beenden }
    .
    .
end else
begin
    .
    .
  { DLL wird aus dem Speicher entfernt }
    .
    .
  end;
  ExitProc := SaveExit;
end;

begin
  .
  .
  { DLL-Initialisierung durchführen }
```

```
  .
  .
  SaveExit := ExitProc; { Zeiger auf vorherige Austrittsprozedur
speichern }
  ExitProc := @LibExit; { Austrittsprozedur LibExit installieren }
end.
```

Wenn Windows eine DLL aus dem Speicher entfernt, wird zunächst in der DLL nach einer exportierten Funktion namens *WEP* gesucht und – wenn gefunden – ausgeführt. Eine Turbo-Pascal-Bibliothek exportiert automatisch eine *WEP*-Funktion. Dabei wird einfach die Adresse, die in der Variablen *ExitProc* gespeichert ist, solange aufgerufen, bis *ExitProc* **nil** wird. Da dies der Behandlung von Exit-Prozeduren in Turbo-Pascal-Programmen entspricht, können Sie dasselbe Austrittsprozedur-Prinzip sowohl bei Programmen als auch bei Bibliotheken verwenden.

Hinweis: Austrittsprozeduren in einer DLL müssen mit ausgeschalteter Stackprüfung (also im Modus **{$S-}**) kompiliert werden, da Windows auf einen internen Stack umschaltet, wenn eine DLL beendet wird. Windows wird außerdem abgebrochen, wenn ein Laufzeitfehler in einer DLL-Austrittsprozedur auftaucht. Sie müssen also entsprechend viele Prüfungen in Ihre Programm einbauen, um Laufzeitfehler zu vermeiden.

Hinweise zur Programmierung einer Bibliothek

Die folgenden Abschnitte enthalten wichtige Fakten, die Sie bei der Arbeit mit DLLs beachten sollten.

Globale Variablen in einer DLL

Jede DLL definiert ihr eigenes Datensegment. Alle Variablen, die in einer DLL deklariert werden, beziehen sich nur auf diese DLL. Eine DLL kann nicht auf Variablen zugreifen, die von anderen Modulen, die die DLL aufrufen, deklariert wurden. Entsprechend können DLLs auch keine Variablen exportieren, die mit anderen Modulen verwendet werden sollen. Solche Zugriffe müssen über eine prozedurale Schnittstelle erfolgen.

Globaler Speicher und Dateien in einer DLL

Grundsätzlich »besitzt« eine DLL nicht die Dateien, die sie öffnet, oder die Speicherblöcke, die sie im System reserviert. Solche Objekte gehören zur Anwendung, die (direkt oder indirekt) die DLL aufgerufen hat. Wenn eine Anwen-

dung beendet wird, werden alle geöffneten Dateien, die zu dieser Anwendung gehören, automatisch geschlossen. Alle globalen Speicherblöcke werden automatisch freigegeben. Das bedeutet, daß Datei- und Speicherblock-Handles, die in den globalen Variablen in einer DLL gespeichert werden, jederzeit ungültig werden können, ohne daß die DLL davon verständigt wird. Aus diesem Grund sollten DLLs keine Annahmen über die Gültigkeit von Datei- und Speicherblock-Handles machen, die in globalen Variablen gespeichert sind. Stattdessen sollten solche Objekt-Handles als Parametern den Prozeduren und Funktionen der DLL übergeben werden. Die Anwendung, die den Aufruf macht, sollte für die Verwaltung verantwortlich sein.

DLLs und die Unit System

Während eine DLL aktiv ist, enthält die Variable *HInstance* die Instanz-Handle der DLL. Die Variablen *HPrevInst* und *CmdShow* sind in einer DLL immer Null. Dasselbe gilt für die Variable *PrefixSeg*, da eine DLL kein Programmsegmentpräfix (PSP) besitzt. In einer Anwendung dagegen ist *PrefixSeg* niemals Null, Daher ergibt der Test *PrefixSeg 0* immer *True*, wenn das aktuelle Modul ein Programm ist, und immer *False*, wenn das aktuelle Modul eine DLL ist.

Weitere Hinweise zum Heap-Manager finden Sie in Kapitel 16 »Speicherbelegung und Datenformate«.

Um die korrekte Funktion des Heap-Managers in der Unit *System* sicherzustellen, setzt der Startcode einer Bibliothek die Variable *HeapLimit* auf Null. Damit wird mit jedem Aufruf zu *New* oder *GetMem* ein eindeutiger globaler Speicherblock reserviert, und die Fähigkeit des Heap-Managers, Teilbereiche eines Speicherblocks zu reservieren und wieder freizugeben, außer Kraft gesetzt. Da jede Speicherreservierung im globalen Heap dazu führt, daß zumindest 32 Bytes belegt werden, sollte eine DLL vermeiden, eine große Anzahl kleiner Speicherblöcke auf dem Heap zu reservieren. Wenn sichergestellt ist, daß eine DLL immer nur von einer Anwendung gleichzeitig verwendet wird, kann die Fähigkeit des Heap-Managers, Teilbereiche eines Speicherblocks zu reservieren, wieder eingeschaltet werden, indem der Standardwert 1024 in der Variable *HeapLimit* gespeichert wird.

Laufzeitfehler in DLLs

Wenn ein Laufzeitfehler in einer DLL auftritt, wird die Anwendung, die die DLL aufgerufen hat, abgebrochen. Die DLL selber wird nicht unbedingt aus dem Speicher entfernt, da sie eventuell noch von einer anderen Anwendung verwendet wird. Da eine DLL selber nicht feststellen kann, ob sie von einer Turbo-Pascal-Anwendung oder einer Anwendung, die in einer anderen Programmiersprache geschrieben wurde, aufgerufen wurde, kann die DLL die Austrittsprozeduren der Anwendung nicht aktivieren, bevor die Anwendung beendet wird. Eine Anwendung wird einfach abgebrochen und aus dem Speicher entfernt. Aus

diesem Grund müssen Sie sicherstellen, daß Ihr Programm solche Fehlerbedingungen abprüft und angemessen darauf reagiert.

DLLs und Stacksegmente

Im Gegensatz zu anderen Anwendungen, hat eine DLL kein eigenes Stacksegment. Sie benutzt das Stacksegment der Anwendung, die die DLL aufgerufen hat. Das kann in DLL-Routinen, die davon ausgehen, daß sich das DS- und das SS-Register auf dasselbe Segment beziehen, zu Problemen führen. Diese Annahme ist zwar für ein Programm richtig. Dennoch erzeugt der Turbo-Pascal-Compiler niemals Code, der voraussetzt, daß DS und SS auf das gleiche Segment verweisen. Auch keine der Routinen der Turbo-Pascal-Laufzeitbibliothek macht diese Annahme. Wenn Sie Assembler-Routinen schreiben, sollten Sie jedoch sicherstellen, daß Ihre Routine nicht davon ausgeht, daß die SS- und DS-Register denselben Wert enthalten.

II. Die Standard-Units

11. Die Unit System

Die Unit *System* ist die Laufzeitbibliothek von Turbo Pascal. Sie stellt Routinen für Ein- und Ausgabe, die Behandlung von Strings und Gleitkommazahlen, Reservierung und Freigabe von Speicher bereit. Diese Unit wird automatisch von jedem Programm und jeder anderen Unit eingebunden. Daher ist es nicht notwendig (und auch nicht erlaubt), *System* in einer **uses**-Anweisung explizit aufzunehmen.

Standardprozeduren und -funktionen

In diesem Abschnitt geben wir eine tabellarische Zusammenfassung der eingebauten Prozeduren und Funktionen von Turbo Pascal, soweit sie nicht im Zusammenhang mit Ein- und Ausgaben stehen, worauf wir erst im folgenden Abschnitt eingehen. Eine ausführliche Beschreibung der verfügbaren Routinen finden Sie in Kapitel 24 »Verzeichnis der Routinen«.

Standardprozeduren und -funktionen sind in der Unit *System* deklariert, die automatisch als äußerster Block in jedes Programm aufgenommen wird. Aus diesem Grund lassen sich die entsprechenden Bezeichner jederzeit innerhalb des Programms »lokal« redeklarieren.

Die Prozeduren Exit, Halt und RunError

Prozedur	Beschreibung
Exit	beendet die Ausführung des momentanen Blocks mit sofortiger Wirkung.
Halt	bricht die Ausführung des Programms ab und führt einen Rücksprung zum Betriebssystem aus.
RunError	bricht die Ausführung des Programms durch Simulation eines Laufzeitfehlers ab und führt einen Rücksprung zum Betriebssystem aus.

Prozeduren zur dynamischen Verwaltung des Speichers

Prozedur	Beschreibung
Dispose	gibt den Speicherplatz einer dynamischen Variablen wieder frei
FreeMem	gibt einen dynamisch belegten Speicherbereich bestimmter Größe wieder frei
GetMem	belegt einen Speicherbereich der angegebenen Größe für eine dynamische Variable und setzt einen Zeiger auf den Anfang dieses Bereichs
New	erzeugt eine dynamische Variable und setzt einen Zeiger auf den Anfang des durch diese Variable belegten Bereichs

Funktionen zur dynamischen Verwaltung des Speichers

Funktion	Beschreibung
MaxAvail	liefert die Größe des größten freien Heap-Bereichs zurück und ermittelt so, wieviele Bytes momentan maximal für eine dynamische Variable belegt werden können
MemAvail	liefert die Gesamtzahl der freien Bytes auf dem Heap zurück

Mit diesen Prozeduren und Funktionen wird der *Heap* verwaltet, d.h. der noch freie Platz im Hauptspeicher bei der Ausführung eines Programms. Eine Besprechung der verwendeten Techniken finden Sie im Abschnitt »Die dynamische Verwaltung des Speichers« in Kapitel 16 »Speicherbelegung und Datenformate«.

Transfer-Funktionen

Funktion	Beschreibung
Chr	liefert ein Zeichen mit der angegebenen Ordinalzahl zurück
Ord	liefert die Ordinalzahl des angegebenen (ordinalen) Wertes
Round	konvertiert einen Realwert in einen *LongInt* und rundet dabei ab oder auf
Trunc	konvertiert einen Realwert in einen *LongInt*, wobei Nachkommastellen abgeschnitten werden

Hinweis: Die im Standard-Pascal definierten Prozeduren **Pack** und **Unpack** sind in Turbo Pascal nicht verfügbar, weil die entsprechenden Prozesse automatisch ausgeführt werden

Arithmetische Funktionen

Funktion	Beschreibung
Abs	liefert den absoluten Wert des Arguments zurück
ArcTan	liefert den Arcustangens des Arguments zurück
Fos	liefert den Cosinus des Arguments zurück
Exp	liefert das Ergebnis der Berechnung $e^{Argument}$ zurück, wobei $e = 2.7182...$
Frac	liefert die Nachkommastellen des Arguments zurück

Funktion	*Beschreibung*
Int	liefert den ganzzahligen Anteil des Arguments zurück
Ln	liefert den natürlichen Logarithmus des Arguments zurück
Pi	liefert den Wert *Pi* (3.141592...) zurück
Sin	liefert den Sinus des Arguments zurück
Sqr	liefert das Quadrat des Arguments zurück
Sqrt	liefert die Quadratwurzel des Arguments zurück

Hinweis: Diese Routinen liefern Werte vom Typ *Real* zurück, wenn ohne Coprozessor bzw. den Emulator gearbeitet wird ({$N-}). Im Modus {$N+} haben die Ergebnisse dagegen den Typ *Extended*:

Ordinale Prozeduren

Prozedur	Beschreibung
Dec	erniedrigt eine Variable
Inc	erhöht eine Variable

Ordinale Funktionen

Funktion	Beschreibung
Odd	prüft, ob es sich bei dem Argument um eine ungerade Zahl handelt, und liefert einen Wahrheitswert zurück
Pred	liefert den Vorgänger des Arguments zurück
Succ	liefert den Nachfolger des Arguments zurück

Stringprozeduren

Prozedur	Beschreibung
Delete	löscht einen Teil eines Strings
Insert	fügt einen Stringteil in einen String ein
Str	konvertiert einen numerischen Wert in einen String
Val	konvertiert einen String in einen numerischen Wert

Stringfunktionen

Funktion	Beschreibung
Concat	verbindet eine Folge von Strings und liefert das Ergebnis zurück
Copy	liefert einen Teil eines Strings zurück
Length	liefert die dynamische Länge eines Strings zurück
Pos	sucht eine Zeichenfolge innerhalb eines Strings und liefert ihre Position zurück

Zeiger- und Adreß-Funktionen

Funktion	Beschreibung
Addr	liefert die Adresse des angegebenen Objekts zurück

Funktion	*Beschreibung*
CSeg	liefert den momentanen Wert des CS-Registers zurück
DSeg	liefert den momentanen Wert des DS-Registers zurück
Ofs	liefert die *Offset*-Adresse des angegebenen Objekts zurück, d.h. die Adresse des Objekts innerhalb eines Segments
Ptr	konvertiert zwei Werte für *Segment* und *Offset* in einen Zeigerwert und liefert das Ergebnis zurück
Seg	liefert die Segment-Adresse des angegebenen Objekts zurück
SPtr	liefert den momentanen Wert des SP-Registers zurück
SSeg	liefert den momentanen Wert des SS-Registers zurück

Verschiedenartige Standardprozeduren

Prozedur	Beschreibung
FillChar	füllt einen Speicherbereich angegebener Größe byteweise mit einem bestimmten Wert
Move	kopiert eine Anzahl von Bytes von einem Speicherbereich in einen anderen
Randomize	initialisiert den eingebauten »Zufallszahlen«-Generator

Verschiedenartige Standardfunktionen

Funktion	Beschreibung
Hi	liefert das höherwertige Byte des Arguments zurück
Lo	liefert das niederwertige Byte des Arguments zurück
ParamCount	liefert die Anzahl der Kommandozeilen-Parameter zurück, mit denen das Programm gestartet wurde
ParamStr	liefert einen Kommandozeilen-Parameter zurück
Random	liefert eine »Zufallszahl« zurück
SizeOf	liefert die Größe des angegebenen Objekts in Bytes zurück
Swap	vertauscht höherwertiges und niederwertiges Byte des Arguments miteinander und liefert das Ergebnis zurück
UpCase	konvertiert ein Zeichen in einen Großbuchstaben

Ein- und Ausgabe

Dieser Abschnitt enthält eine kurze Beschreibung der Prozeduren und Funktionen zur Ein- und Ausgabe (I/O), die in Turbo Pascal eingebaut sind. Weitere Informationen finden Sie in Kapitel 19 »Ein- und Ausgabe«.

Grund-legendes

Variablen des Typs *File* werden als Datei-Variablen bezeichnet, wobei zwischen drei Arten unterschieden wird, nämlich *Textdateien*, *typisierten* und *untypisierten* Dateien. Eine Datei-Variable, die als identisch zum Standardtyp *Text* deklariert worden ist, bezeichnet man als Textdatei. Wird eine Datei-Variable über das Konstrukt **file of** <*Komponententyp*> deklariert, handelt es sich um eine typisierte Datei; eine Deklaration als **file** (ohne Angabe eines Komponententyps) erzeugt eine untypisierte Datei-Variable.

Bevor eine Datei-Variable benutzt werden kann, muß sie einer externen Datei durch den Aufruf der Prozedur *Assign* zugeordnet werden. Bei externen Dateien handelt es sich normalerweise um benannte Disketten-Dateien - *Geräte* wie die Tastatur oder der Bildschirm lassen sich aber ebenfalls als externe Datei behandeln.

Der nächste Schritt nach der Herstellung einer Verbindung zwischen der Datei-Variablen und der externen Datei besteht im *Eröffnen* der Datei, d.h. einigen vorbereitenden Schritten für nachfolgende Ein- und/oder Ausgaben. Eine bereits bestehende Datei kann über die Prozedur *Reset* geöffnet werden, die Prozedur *Rewrite* erzeugt und öffnet dagegen neue Dateien. Mit *Reset* geöffnete Textdateien können nur gelesen, mit *Rewrite* geöffnete Dateien nur beschrieben werden. Typisierte und untypisierte Dateien können sowohl gelesen als auch beschrieben werden, wobei es keine Rolle spielt, ob sie mit *Reset* oder *Rewrite* eröffnet worden sind.

Eine Datei besteht aus einer linearen Folge von Komponenten, von denen jede den Komponententyp der Datei hat. Jeder Komponente wird eine *Komponenten-Nummer* zugeordnet, wobei die erste Komponente der Datei die Nummer 0 erhält.

Auf Dateiinhalte wird normalerweise in sequentieller Folge zugegriffen. Durch das Lesen einer Komponente mit der Standardprozedur *Read* oder das Schreiben mit der Standardprozedur *Write* wird die momentane Position in der Datei auf die nächstfolgende Dateikomponente gesetzt. Auf typisierte und untypisierte Dateien kann dagegen auch nach anderen Kriterien zugegriffen werden: Die Standardprozedur *Seek* setzt die momentane Position in der Datei auf die Komponente der dabei angegebenen Nummer; die Standardprozedur *FilePos* bestimmt die momentane Position in der Datei, über die Standardprozedur *FileSize* kann die momentane Größe (d.h. Komponentenzahl) einer Datei ermittelt werden.

Wenn ein Programm die Bearbeitung einer Datei beendet hat, muß die Datei-Variable mit *Close* wieder *geschlossen* werden. Das Schließen einer Datei-Variablen hat eine Aktualisierung der damit verbundenen externen Datei und eventueller Inhaltsverzeichnisse zur Folge. Danach wird die Verbindung zwischen der Datei-Variablen und der externen Datei gelöst, und die Datei-Variable kann einer anderen externen Datei zugeordnet werden.

Das Laufzeitsystem von Turbo Pascal prüft standardmäßig nach jeder Ein-/Ausgabeoperation auf Fehler und bricht gegebenenfalls mit einer Meldung ab. Diese automatische Überprüfung kann mit den Compiler-Befehlen **{$I+}** und

{$I-} an- bzw. abgeschaltet werden. Bei abgeschalteter Überprüfung hat ein Fehler keinen automatischen Abbruch zur Folge, seine Art läßt sich über die Standardfunktion *IOResult* ermitteln.

Standardprozeduren für alle Dateien

Prozedur	Beschreibung
Assign	ordnet einer Datei-Variablen eine externe Datei über ihren Namen zu
ChDir	wechselt das momentan gesetzte Verzeichnis
Close	schließt eine offene Datei
Erase	löscht eine externe Datei
GetDir	ermittelt das momentan gesetzte Verzeichnis des angegebenen Laufwerks
MkDir	erzeugt ein Unterverzeichnis
Rename	gibt einer Datei einen neuen Namen
Reset	öffnet eine existierende Datei
Rewrite	erzeugt eine Datei und öffnet sie
RmDir	löscht ein (leeres) Unterverzeichnis
Seek	setzt eine Position innerhalb der angegebenen Datei Nicht anwendbar auf Dateien des Typs *Text*.
Truncate	schneidet eine Datei an der momentanen Position ab Nicht anwendbar auf Dateien des Typs *Text*.

Standardfunktionen für alle Dateien

Funktion	Beschreibung
Eof	prüft, ob das Ende der angegebenen Datei erreicht ist, und liefert einen Wahrheitswert zurück
FilePos	liefert die momentane Position innerhalb einer Datei zurück. Nicht anwendbar auf Dateien des Typs *Text*.
FileSize	liefert die Größe der angegebenen Datei zurück Nicht anwendbar auf Dateien des Typs *Text*.
IOResult	liefert den Status der zuletzt ausgeführten I/O-Operation als Integerwert zurück

Die beiden Tabellen geben einen kurzen Überblick über die eingebauten Funktionen und Prozeduren von Turbo Pascal, die auf Dateien aller drei Arten angewendet werden können.

Standard-Prozeduren und Funktionen für Textdateien

Die in diesem Abschnitt beschriebenen Funktionen und Prozeduren sind nur auf Dateien des Typs *Text* anwendbar. Bitte beachten Sie, daß Turbo Pascal zwischen den Datei-Variablentypen *Text* und *File of Char* unterscheidet; im letzteren Fall gilt die Datei als typisiert.

Textdateien werden beim Öffnen auf besondere Art und Weise interpretiert,

nämlich als Folge von zeilenweise angeordneten Zeichen, wobei jede Zeile durch ein *Zeilenendezeichen* abgeschlossen ist ([–]-Zeichen, ASCII-Code 13, danach eventuell ein Zeilenvorschub (LF), ASCII-Code 10).

Für Textdateien existieren Sonderformen von *Read* und *Write*, die auch das Schreiben und Lesen von Werten ermöglichen, die nicht den Typ *Char* haben. Strings werden als Folge einzelner Zeichen geschrieben und beim Lesen wieder in Strings gespeichert; numerische Werte werden als Folgen numerischer Zeichen geschrieben und beim Lesen zurückverwandelt. Wenn *i* eine Integer-Variable ist, dann liest die Anweisung *Read(f,i)* eine Folge von Ziffern aus der durch *f* bezeichneten Datei, interpretiert sie als dezimalen Integerwert und speichert diesen Wert schließlich in *i*.

Turbo Pascal deklariert die Standard-Dateien *Input* und *Output* automatisch. Beide haben den Typ *Text*, wobei *Input* normalerweise der Tastatur zugeordnet ist und nur gelesen werden kann, *Output* ist normalerweise dem Bildschirm zugeordnet und kann nur beschrieben werden.

Da Windows die textorientierte Ein- und Ausgabe nicht direkt unterstützt, öffnet Turbo Pascal in einem Windows-Programm die Dateien *Input* und *Output* nicht automatisch. Daher wird jeder Versuch, aus diesen Dateien zu lesen oder in sie zu schreiben, mit einem Ein-/Ausgabefehler quittiert. Bindet jedoch ein Programm die Unit *WinCrt* ein, beziehen sich diese Dateien auf ein Fenster, das unter Windows ein Textfenster simuliert. Alles, was zur Verwaltung dieses »Textfensters« notwendig ist, definiert die Unit *WinCrt*. Weitere, Windows-spezifische Programmkunststücke sind nicht erforderlich.

Sämtliche der im folgenden beschriebenen Prozeduren und Funktionen arbeiten mit einer externen Textdatei, wenn eine Datei-Variable angegeben ist – ansonsten beziehen sie sich auf Tastatur bzw. Bildschirm. Folglich hat die Anweisung *Read(x)* dieselbe Bedeutung (und Wirkung) wie *Read(Input, x)*; Write(x) und *Write(Output, x)* sind ebenfalls identisch.

Bevor eine dieser Prozeduren und Funktionen auf eine externe Datei angewendet wird, muß diese Datei mit *Assign* zugeordnet und mit *Reset*, *Rewrite* oder *Append* geöffnet werden. Der Versuch, aus einer mit *Rewrite* oder *Append* geöffneten Textdatei etwas zu lesen oder in eine mit *Reset* geöffnete Textdatei etwas hineinzuschreiben, erzeugt einen Laufzeitfehler.

Standardfunktionen

Funktion	Beschreibung
Eoln	prüft, ob in der angegebenen Datei ein Zeilenende erreicht wurde, und liefert einen Wahrheitswert zurück
SeekEof	prüft, ob das Ende der angegebenen Datei erreicht werden kann, und liefert einen Wahrheitswert zurück
SeekEoln	prüft, ob in der angegebenen Datei ein Zeilenende erreicht werden kann, und liefert einen Wahrheitswert zurück

Standardprozeduren

Prozedur	Beschreibung
Append	öffnet eine existierende Datei für nachfolgende Schreiboperationen, neue Daten werden »angehängt«
Flush	erzwingt das physikalische Schreiben eines Ausgabepuffers
Read	liest einen oder mehrere Werte aus der angegebenen Textdatei in die entsprechende Anzahl von Variablen
Readln	hat dieselbe Funktion wie *Read*, springt aber danach bis zum Anfang der nächsten Zeile in der Textdatei
SetTextBuf	ordnet einer Textdatei einen Ein-/Ausgabepuffer zu
Write	schreibt einen oder mehrere Werte in eine Textdatei
Writeln	hat dieselbe Funktion wie *Write*, schreibt aber danach automatisch ein Zeilenendezeichen

Standard-Prozeduren für untypisierte Dateien

Untypisierte Dateien werden primär für Zugriffe auf niedriger Ebene benutzt, d.h. Zugriffe, bei denen keine Voraussetzungen für Struktur und Inhalt der entsprechenden Disketten-Dateien gemacht werden. Die entsprechenden Datei-Variablen werden durch das reservierte Wort **file** (ohne weitere Zusätze) deklariert:

```
var
  Binaerdatei: file;
```

Bei der Anwendung auf untypisierte Dateien erlauben die Prozeduren *Reset* und *Rewrite* einen weiteren Parameter zur Festlegung der Recordgröße. Die Recordgröße bestimmt, wieviele Bytes bei einem Zugriff gelesen bzw. geschrieben werden; sie wird aus »historischen« Gründen mit 128 angenommen, wenn keine explizite Angabe erfolgt. Vorzuziehen ist allerdings eine Größe von 1, weil nur dadurch der exakte Umfang der Datei bestimmt werden kann (bei allen anderen Größen wäre ein nur teilweise gefüllter Record am Dateiende möglich).

Abgesehen von *Read*, *Readln*, *Write* und *Writeln* sind alle Standard-Dateiprozeduren und -funktionen von Turbo Pascal auch auf untypisierte Dateien anwendbar. *Read* und *Write* werden durch *BlockRead* und *BlockWrite* ersetzt - zwei Prozeduren, die Block-Datentransfers mit hoher Geschwindigkeit ermöglichen.

Standardprozeduren

Prozedur	Beschreibung
BlockRead	liest einen oder mehrere Records aus der angegebenen Datei in eine Variable
BlockWrite	schreibt einen oder mehrere Records aus einer Variablen in die angegebene Datei

Die Variable FileMode

ist in der Unit *System* deklariert – ihr Wert legt fest, in welchem Modus typisierte und untypisierte Dateien durch *Reset* geöffnet werden. (Dateien des Typs *Text* werden durch *Reset* grundsätzlich für Leseaktionen geöffnet - hier hat *FileMode* keinen Einfluß).

Beim Betrieb unter DOS 2.x kann *FileMode* einem von drei verschiedenen Werten zugewiesen werden:

0	Öffnen von untypisierten Dateien nur für Leseaktionen
1	nur für Schreibaktionen
2	für Lese- und Schreibaktionen gleichzeitig

Die Standardvorgabe für *FileMode* ist der Wert 2 - typisierte und untypisierte Dateien werden von *Reset* also für Lese- und Schreibaktionen gleichzeitig geöffnet.

Die DOS-Versionen 3.x definieren einige weitere Modi für die Bearbeitung von Dateien in Netzen (und wir verweisen an dieser Stelle auf das DOS-Handbuch).

Hinweis: Die Prozedur *Rewrite* öffnet typisierte und untypisierte Dateien grundsätzlich für Lese- und Schreibaktionen gleichzeitig, arbeitet also mit dem Äquivalent von *FileMode = 2*. (Der aktuelle Wert von *FileMode* wird durch *Rewrite* nicht verändert).

Geräte in Turbo Pascal

Peripheriegeräte werden in Turbo Pascal unter dem Sammelbegriff *Geräte* behandelt; dabei kann es sich um die Tastatur, den Bildschirm, den Drucker oder andere angeschlossene Elektronik handeln. Aus der Sicht des Programmierers werden Geräte als Datei behandelt, wobei dieselben Funktionen und Prozeduren wie bei Diskettendateien zur Anwendung kommen.

Abhängig von der Natur angeschlossener Peripheriegeräte unterscheidet Turbo Pascal zwischen den Geräteklassen »DOS« und »Text«.

DOS-Geräteeinheiten

sind über eine Reihe spezieller (und für diesen Zweck reservierter) Dateinamen implementiert. Die zugrundeliegenden Mechanismen von DOS arbeiten unsichtbar - es macht auch technisch keinen Unterschied, ob sich eine Datei-Variable auf ein DOS-Gerät oder eine Diskettendatei bezieht. Ein Programm wie

```
var
  Lst: Text;

begin
  Assign(Lst,'LPT1'); Rewrite(Lst);
```

```
  Writeln(Lst, 'Eine Ausgabe');
  Close(Lst);
end.
```

gibt die Zeile `Eine Ausgabe` auf dem Drucker aus - obwohl die verwendete Syntax exakt dieselbe ist wie bei der Ausgabe in eine Diskettendatei.

Die durch DOS implementierten Geräte arbeiten in den allermeisten Fällen mit »lesbaren« Ein- und Ausgaben, d.h. werden in Verbindung mit Textdateien benutzt. (Für wenige und spezialisierte Aufgabenstellungen lassen sich auch untypisierte Dateien verwenden).

Im allgemeinen sollten Sie unter Windows DOS-Geräteeinheiten nicht direkt ansprechen und stattdessen die zur Ein- und Ausgabe vorgesehenen Routinen des Windows-API aufrufen. Dennoch funktioniert bei einigen DOS-Geräteeinheiten (z.B. LPT1) das oben beschriebene Verfahren, während es bei anderen (z.B. CON) fehlschlägt.

Text-Geräteeinheiten

werden verwendet, um von DOS nicht unterstützte Geräte zu implementieren oder vorhandene Geräte zu ersetzen. Die Standard-Unit *WinCrt* ist ein gutes Beispiel dafür: sie simuliert unter Windows ein »Textfenster«, und erlaubt Ihnen, ohne größere Anstrengungen ein Programm zu schreiben, das die Standardroutinen zur Ein- und Ausgabe verwendet.

Geräteeinheiten der Klasse »Text« verwenden (im Gegensatz zu »DOS«) keine reservierten Dateinamen - sie arbeiten über *Gerätetreiber* und werden über spezielle Versionen von *Assign* zugeordnet. Die Unit *WinCrt* arbeitet beispielsweise mit der (selbstdefinierten) Prozedur *AssignCrt*.

Vordefinierte Variablen

Neben Prozeduren und Funktionen deklariert die Unit *System* eine Reihe von Variablen und typsierten Konstanten:

Variable	Typ	Beschreibung
Input	Text	Standard-Eingabe
Output	Text	Standard-Ausgabe

Wie oben schon erwähnt, öffnet Turbo Pascal für Windows diese Dateien nicht automatisch beim Start eines Programms.

Variable	Typ	Anfangswert	Beschreibung
CmdLine	PChar	**nil;**	Zeiger auf die Kommandozeile
CmdShow	Integer	0	Parameter für CreateWindow
ErrorAddr	Pointer	**nil**	Adresse eines evtl. Laufzeitfehlers

Variable	*Typ*	*Anfangs-wert*	*Beschreibung*
ExitCode	Integer	0	Exit-Code des Programms
ExitProc	Pointer	**nil**	zuletzt eingesetzte Exit-Prozedur
FileMode	Byte	2	Standardmodus für das Öffnen von Dateien
HeapBlock	Word	8192	Größe eines Heap-Speicherblocks
HeapError	Pointer	**nil**	für eigene Fehlerbehandlungsroutinen
HeapLimit	Word	1024	»kleiner« Heap-Speicherblock
HeapList	Word	0	Heap-Segmentliste
HInstance	Word	0	Handle aktuelle Instanz
HPrevInst	Word	0	Handle vorige Instanz
InOutRes	Integer	0	Status für IOResult
PrefixSeg	Word	0	PSP-Segmentadresse
RandSeed	LongInt	0	Startwert des »Zufallszahlen« Generators

HInstance speichert das von Windows gelieferte Instanz-Handle des Programms oder der Bibliothek. In einem Programm enthält *HPrevInst* den Handle der vorhergehenden Instanz der Applikation oder 0, wenn eine solche Instanz nicht existiert. In einer Bibliothek ist *HPrevInst* immer auf 0 gesetzt.

In einem Programm zeigt *CmdLine* auf einem null-terminierten String, der die Parameter enthält, die dem Programm bei seinem Start übergeben wurden. In einer Bibliothek ist *CmdLine* undefiniert.

In einem Programm speichert *CmdShow* den Parameterwert, den Window an *ShowWindow* übergibt, sobald das Programm sein Fenster aufbaut. In einer Bibliothek ist *CmdShow* immer auf 0 gesetzt.

Org (»Heap Origin« = Beginn des Heap), *HeapPtr*, *HeapEnd*, *FreeList*, *FreeZero* und *HeapError* werden von den Routinen zur dynamischen Verwaltung des Speichers benutzt - wir gehen in Kapitel 14 »Interne Details« ausführlich darauf ein.

Der Heap-Manager verwendet *HeapList*, *HeapLimit*, *HeapBlock* und *HeapError* bei der Belegung und Freigabe dynamischen Speichers. Einzelheiten dazu finden Sie im Kapitel 18 »Interne Details.«.

ExitProc ist eine Zeigervariable, über die eine Prozedur zum Beenden des Programms festgelegt werden kann; *ExitCode* hält den vom Programm zurückgelieferten Exit-Code fest, *ErrorAddr* die Adresse eines eventuellen Laufzeitfehlers. Auf diese drei Variablen gehen wir im Zusammenhang mit Exit-Prozeduren ebenfalls in Kapitel 18 »Interne Details« ausführlicher ein.

In einem Programm enthält *PrefixSeg* die Segmentadresse des *Programmsegment-Präfix* (PSP) – eines Speicherbereichs, den DOS und Windows jedem Programm automatisch voranstellen. In einer Bibliothek ist *PrefixSeg* immer auf 0 gesetzt. Eine detaillierte Beschreibung der in diesem Präfix gespeicherten Daten finden Sie in den DOS- bzw. Windows-Handbüchern.

Die Ein-/Ausgaberoutinen speichern in *InOutRes* den Wert, der von der Standardroutine *IOResult* zurückgegeben wird.

RandSeed liefert den Startwert des eingebauten »Zufallszahlen«-Generators –

durch direkte Zuweisungen können über *Random* erzeugte Zahlenfolgen beliebig oft wiederholt werden. Diese Möglichkeit ist hauptsächlich für das Austesten von Programmen sowie Verschlüsselungen, Statistiken und Simulationen gedacht.

12. Die Unit WinDos

Die Unit *WinDos* implementiert einige Betriebssystem- und Dateihandhabungs-Routinen. Da keine dieser Routinen in der Unit *WinDos* in Standard-Pascal definiert ist, wurden sie in einer separaten Unit zusammengefaßt. Eine umfassende Beschreibung der DOS-Funktionen finden Sie zum Beispiel im technischen Handbuch zu DOS.

Konstanten, Typen und Variablen

Die folgenden Abschnitte geben einen kurzen Abriß der in der Unit *WinDos* deklarierten Konstanten, Typen und Variablen, wobei nach logischen Funktionsgruppen geordnet wurde. Eine ausführlichere Besprechung der in dieser Unit deklarierten Prozeduren und Funktionen finden Sie in Kapitel 24 »Verzeichnis der Routinen«.

Flag-Konstanten
sind für die Auswertung der einzelnen Bits im Flag-Register des Prozessors nach dem Aufruf der Routinen *Intr* und *MsDos* vorgesehen. Die folgenden Konstanten sind zu diesem Zweck in der Unit *WinDos* definiert:

Konstante	Wert
fCarry	$0001;
fParity	$0004;
fAuxiliary	$0010;
fZero	$0040;
fSign	$0080;
fOverflow	$0800;

Wenn wir beispielsweise annehmen, daß *R* ein Record des Typs *TRegisters* ist, dann liefern die folgenden Tests beide das Ergebnis *True*, wenn das Carry-Flag gesetzt und das Zero-Flag gelöscht ist:

```
R.Flags and fCarry <> 0
R.Flags and fZero = 0
```

Modi der Dateibearbeitung

Die folgenden Konstanten werden von den Routinen zum Öffnen und Schließen von Diskettendateien verwendet. Das Modus-Feld einer Datei-Variablen von Turbo Pascal enthält immer einen der folgenden Werte:

Konstante	Wert
fmClosed	$D7B0
fmInput	$D7B1
fmOutput	$D7B2
fmInOut	$D7B3

Dateiattribut-Konstanten

werden zum Setzen, Testen und Löschen der Attribut-Bits von Dateien über die Prozeduren *GetFAttr*, *SetFAttr*, *FindFirst* und *FindNext* verwendet:

Konstante	Wert
faReadOnly	$01;
faHidden	$02;
faSysFile	$04;
faVolumeID	$08;
faDirectory	$10;
faArchive	$20;
faAnyFile	$3F;

Diese Konstanten sind additiv – ein Aufruf wie

```
FindFirst('*.*', faReadOnly + faDirectory, S);
```

sucht beispielsweise das momentan gesetzte Verzeichnis nach schreibgeschützten Dateien *und* Unterverzeichnissen ab. Die Konstante *faAnyFile* steht für eine Datei mit beliebigen Attributen.

Stringlänge der Komponenten eines Dateinamens

Diese Konstanten legen die maximale Längen der einzelnen Komponenten eines Dateinamens fest, und werden zum Beispiel von den Funktionen *FileSearch* und *FileExpand* verwendet.

Konstante	Wert
fsPathName	79
fsDirectory	67
fsFileName	8
fsExtension	4

Return-Flags für FileSplit

Diese Return-Flags werden von der Funktion *FileSplit* verwendet. Der gelieferte Wert ist eine Kombination aus *fcDirectory-, fcFileName-* und *fcExtension*-Bitmasken, die die im Pfad vorhandenen Komponenten angeben. Wenn der Name oder die Dateinamenserweiterung Jokerzeichen enthält (* oder ?) wird das Flag *fcWildcards* im gelieferten Wert gesetzt.

Konstante	Wert
fcExtension	$0001
fcFileName	$0002
fcDirectory	$0004
fcWildcards	$0008

Typen

Datei-Recordtypen

Die intern von Turbo Pascal verwendeten Record-Typen zur Speicherung von Datei-Variablen sind ebenfalls in der Unit *WinDos* definiert. Der Typ *TFileRec* wird für typisierte und untypisierte Dateien verwendet. *TTextRec* ist das interne Format einer Variable vom Typ *Text*.

```
type
{ typisierte / untypisierte Dateien }
  TFileRec = record
    Handle: Word;
    Mode: Word;
    RecSize: Word;
    Private: array[1..26] of Byte;
    UserData: array[1..16] of Byte;
    Name: array[0..79] of Char;
  end;

{ TextFile Record }
  PTextBuf = ^TTextBuf;
  PTextBuf = array[0..127] of Char;
  PTextRec = record
    Handle: Word;
    Mode: Word;
    BufSize: Word;
    Private: Word;
    BufPos: Word;
    BufEnd: Word;
    BufPtr: ^TTextBuf;
    OpenFunc: Pointer;
    InOutFunc: Pointer;
    FlushFunc: Pointer;
    CloseFunc: Pointer;
    UserData:  array[1..16] of Byte;
    Name: array[0..79] of Char;
    Buffer: TTextBuf;
  end;
```

Der Recordtyp TRegisters
wird von den Prozeduren *Intr* und *MsDos* zur Übergabe der Prozessor-Register verwendet: vor dem Aufruf werden die Register mit den im Record enthaltenen Feldern gesetzt, nach dem Aufruf werden die Ergebnisse in diesem Record gespeichert.

```
type
  TRegisters = record
    case Integer of
    0: (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Word);
    1: (AL,AH,BL,BH,CL,CH,DL,DH: Byte);
  end;
```

TRegisters als varianter Record definiert; die 8-Bit-Register belegen denselben Speicherplatz wie ihre 16-Bit-Äquivalente.

Der Recordtyp TDateTime
wird zusammen mit den Prozeduren *PackTime* und *UnpackTime* verwendet, die Datum und Uhrzeit in ein gepacktes Format (4 Byte) konvertieren. Die Prozeduren *GetFTime*, *SetFTime*, *FindFirst* und *FindNext* arbeiten mit Daten dieses gepackten Formats.

```
type
  TDateTime = record
    Year,Month,Day,Hour,Min,Sec: Word;
  end;
```

Die zulässigen Bereiche für *Month, Day*, *Hour* gehen von 1..12, 1..31 und 0..23, für *Min* und *Sec* jeweils von 0..59. Mögliche Werte für das Feld *Year* sind auf den Bereich 1980..2099 beschränkt.

Der Recordtyp TSearchRec
wird ausschließlich von den Prozeduren *FindFirst* und *FindNext* verwendet, die ein Verzeichnis nach dem ersten bzw. jeweils nächsten Vorkommen eines Dateieintrags absuchen.

```
type
  TSearchRec = record
    Fill: array[1..21] of Byte;
    Attr: Byte;
    Time: Longint;
    Size: Longint;
    Name: string[12];
  end;
```

Beide Prozeduren verwenden einen Record diesen Typs, um die Daten eines gefundenen Dateieintrags zu speichern. Das Feld *Attr* enthält die Attribute der Datei, *Time* das Datum der Erstellung (um hier den »Klartext« zu erhalten, müssen Sie *UnpackTime* anwenden). *Size* enthält die Größe der Datei in Bytes, *Name* den Namen. Das Feld *Fill* wird von DOS zur Speicherung interner Daten

(und für die nächste Suche via *FindNext*) verwendet – seinen Inhalt sollte man nie verändern.

Variablen

Die Variable DosError
wird von den meisten Routinen in der Unit *WinDos* für die Übergabe von Fehlercodes verwendet. Ihre Deklaration:

```
var   DosError: Integer;
```

Die hier gespeicherten Werte entsprechen direkt den Fehlercodes von DOS. 0 steht für »fehlerfreie Ausführung«, andere mögliche Werte und ihre Bedeutung finden Sie in der folgenden Tabelle:

DOS-Fehlercode	Bedeutung
2	File not found (Datei nicht gefunden)
3	Path not found (Suchweg nicht gefunden)
5	Access denied (Zugriff verweigert)
6	Invalid handle (Handle nicht definiert/ungültig)
8	Not enough memory (nicht genügend Platz im Hauptspeicher)
10	Invalid environment (»Umgebungs«-Parameter ungültig)
11	Invalid format (ungültiges Befehlsformat)
18	No more files (keine weiteren Dateieinträge)

Prozeduren und Funktionen

Prozeduren für Datum und Uhrzeit

Prozedur	Beschreibung
GetDate	ermittelt das momentane Kalenderdatum des Systems
GetFTime	ermittelt Datum und Uhrzeit der letzten Veränderung einer offenen Datei
GetTime	ermittelt die momentan gesetzte Uhrzeit des Systems
PackTime	konvertiert einen Record des Typs *TDateTime* in eine Struktur von 4 Bytes (formal: Typ *LongInt*) für *SetFTime*
SetDate	setzt das Kalenderdatum des Systems
SetFTime	setzt Modifikations-Datum und Uhrzeit einer offenen Datei
SetTime	setzt die Uhrzeit des Systems
UnpackTime	konvertiert eine gepackte Struktur von 4 Bytes (formal: Typ *LongInt*) in einen Record des Typs *TDateTime*.

Derartige Strukturen werden von *GetFTime*, *FindFirst* und *FindNext* erzeugt

Interrupt-Support-Prozeduren

Prozedur	Beschreibung
GetIntVec	ermittelt die Adresse, auf die ein Interrupt-Vektor zeigt
Intr	führt einen Software-Interrupt aus
MsDos	führt einen DOS-Funktionsaufruf aus
SetIntVec	setzt einen Interrupt-Vektor auf eine bestimmte Adresse

Statusfunktionen für Disketten- und Festplattenlaufwerke

Funktion	Beschreibung
DiskFree	liefert die Anzahl der freien Bytes auf dem angegebenen Laufwerk zurück
DiskSize	liefert die Gesamtgröße des Datenträgers im angegebenen Laufwerk zurück

Prozeduren zur Bearbeitung von Dateien

Prozedur	Beschreibung
FindFirst	sucht das angegebene bzw. momentan gesetzte Verzeichnis nach dem ersten Eintrag ab, der einem bestimmten Dateinamen und festgelegten Attributen entspricht
FindNext	setzt eine mit *FindFirst* begonnene Suche fort
GetFAttr	ermittelt die Attribute einer Datei
SetFAttr	setzt die Attribute einer Datei

Funktionen zur Bearbeitung von Dateien

Funktion	Beschreibung
FileSplit	zerlegt einen Dateinamen in seine drei Komponenten »Suchweg«, »Name« und »Suffix«
FileExpand	erweitert einen Dateinamen um den Suchweg des momentan gesetzten Verzeichnisses
FileSearch	sucht Verzeichnis-Listen nach Dateien ab

Prozeduren zur Bearbeitung von Verzeichnissen

Prozedur	Beschreibung
CreateDir	erzeugt ein neues Unterverzeichnis
RemoveDir	entfernt ein Unterverzeichnis
SetCurDir	wechselt das aktuelle Verzeichnis

Funktionen zur Bearbeitung von Verzeichnissen

Funktion	Beschreibung
GetCurDir	liefert das aktuelle Verzeichnis im angegebenen Laufwerk

Funktionen zur Bearbeitung von Environmenteinträgen

Funktion	Beschreibung
GetArgCount	liefert die Anzahl der Parameter, die dem Programm in der Kommandozeile übergeben wurden
GetArgStr	liefert den angegebenen Environment-String
GetEnvVar	liefert einen Zeiger auf den Wert einer angegebenen Environment-Variablen

Verschiedene Prozeduren

Prozedur	Beschreibung
GetCBreak	ermittelt den momentanen Status von *Ctrl-Break* von DOS
GetVerify	ermittelt den momentanen Stand des *Verify*-Flags von DOS
SetCBreak	setzt den Status für *Ctrl-Break* von DOS
SetVerify	setzt das *Verify*-Flag von DOS

Verschiedene Funktionen

Funktion	Beschreibung
DosVersion	liefert die Versionsnummer von DOS

13. Die Unit Strings

Turbo Pascal für Windows definiert eine neue Klasse von Zeichenstrings, die sogenannten *null-terminierten Strings*. Turbo Pascals erweiterte Syntax zusammen mit der Unit *Strings* ermöglichen Ihrem Programmen die Verwendung null-terminierter Strings. Dieses Format erwartet auch das Windows-API (Application Programming Interface).

Was ist ein null-terminierter String?

Der Compiler speichert einen traditionellen Pascal-String als ein Längen-Byte gefolgt von einer entsprechend langen Sequenz von Zeichen. Damit kann ein Turbo-Pascal-String maximal 255 Zeichen lang sein. Er belegt zwischen 1 und 256 Byte Speicher.

Ein null-terminierter String hat kein Längen-Byte. Statt dessen besteht er aus einer Folge von Nicht-Null-Zeichen gefolgt von einem NULL(#0)-Zeichen. Damit ist ein solcher String auf keine vorgegebene Maximallänge beschränkt. Durch die 16-Bit-Architektur von DOS und Windows ergibt sich jedoch die obere Grenze von 65535 Zeichen.

Mit *StrPCopy* kann ein Pascal-String in einen null-terminierten String kopiert werden. Mit *StrPas* kann ein null-terminierter String zu einem Pascal-String konvertiert werden.

Verwendung von null-terminierten Strings

Null-terminierte Strings werden als Arrays, deren Indexzählung mit Null beginnt, gespeichert. Ein solches Array hat damit folgenden Aufbau

```
array [0..X] of Char
```

wobei X ein positiver Integer ungleich Null ist. Im folgenden finden Sie einige Beispiele für Zeichen-Arrays, die zur Speicherung von null-terminierten Strings verwendet werden können.

```
type
  TIdentifier = array [0..15] of Char;
  TFileName = array [0..79] of Char;
  TMemoText = array [0..1023] of Char;
```

Der größte Unterschied zwischen Pascal-Strings und null-terminierten Strings ist die sehr häufige Verwendung von *Zeigern* zur Manipulation von null-terminierten Strings. Für Operationen mit diesen Zeigern definiert Turbo Pascal für Windows eine Gruppe *erweiterter Syntax-Regeln* zusammen mit dem neuen Datentyp *PChar*, der einen Zeiger auf einen null-terminierten String darstellt. Die Unit *System* deklariert *PChar* als

```
type PChar = ^Char;
```

Der Compilerbefehl **$X** aktiviert oder deaktiviert die Gültigkeit der erweiterten Syntax-Regeln. Wenn **$X** auf **{$X+}** (die Voreinstellung) gesetzt ist, ist diese erweiterte Turbo-Pascal-Syntax eingeschaltet. Der folgende Abschnitt gibt einen Überblick über die erweiterte Syntax.

Zeichenzeiger und String-Literale

Bei eingeschalteter erweiterter Syntax ist ein String-Literal zum Typ *PChar* zuweisungskompatibel. Das bedeutet, daß einer Variablen vom Typ *PChar* ein String-Literal zugewiesen werden kann. Zum Beispiel

```
var
  P: PChar;
  .
  .
begin
  P := 'Hallo Erdling....';
end;
```

Das Ergebnis einer solchen Zuweisung ist, daß der Zeiger auf einen Speicherbereich zeigt, der eine null-terminierte Kopie des String-Literals enthält. Der Compiler speichert die String-Literale im Datensegment, ähnlich einer »versteckten« typisierten Konstantendeklaration:

```
const
  TempString: array [0..14] of Char = 'Hallo Erdling...'#0;
var
 P: PChar;
 .
 .
begin
 P := @TempString;
end;
```

String-Literale können in Prozedur- und Funktionsaufrufen als Parameter verwendet werden, wenn der entsprechende formale Parameter den Typ *PChar* hat. Für folgende Prozedur mit der Deklaration

```
procedure PrintStr(Str: PChar);
```

sind die folgenden Prozeduraufrufe gültig:

```
PrintStr ('Das ist ein Test');
PrintStr (#10#13);
```

Genau wie bei einer Anweisung erzeugt der Compiler eine null-terminierte Kopie des String-Literals im Datensegment und übergibt im Parameter *Str* der Prozedur *PrintStr* einen Zeiger auf diesen Speicherbereich.

Schließlich kann eine typisierte Konstante des Typs *PChar* mit einer String-Konstanten initialisiert werden. Dies ist auch möglich bei strukturierten Typen, wie zum Beispiel Arrays von *PChar* und Records und Objekte mit *PChar*-Feldern.

```
const
  Message: PChar = 'Programm abgebrochen';
  Prompt: PChar = 'Werte eingeben:';
  Digits: array [0..9] of PChar = (
    'Null', 'Eins', 'Zwei', 'Drei', 'Vier',
    'Fünf', 'Sechs', 'Sieben', 'Acht', 'Neun');
```

Zeiger auf Zeichen und Zeichenarrays

Bei eingeschalteter erweiterter Syntax (mit **$X+**), ist ein Array, dessen Indexzählung mit Null beginnt, mit dem Typ PChar *kompatibel*. Das bedeutet, daß immer wenn *PChar* erwartet wird, stattdessen auch ein solches Array verwendet werden kann. Ist das der Fall, wandelt der Compiler das Zeichenarray in eine *Zeigerkonstante* um, deren Wert der Adresse des ersten Elements des Arrays entspricht.

```
var
  A: array[0..63] of Char;
  P: PChar
  .
  .
begin
  P := A;
  PrintStr(A);
  PrintStr(P);
end;
```

Wegen dieser Zuweisung zeigt *P* nun auf das erste Element von *A*, so daß *PrintStr* zweimal mit dem selben Wert aufgerufen wird.

Eine typisierte Konstante, die als **array**[0..X] of Char definiert ist, kann mit einem String-Literal initialisiert werden, das kürzer als die deklarierte Länge des

Arrays ist. Die übrigen Zeichen werden auf NULL (#0) gesetzt, wodurch das Array einen null-terminierten String speichert.

```
type
  TFileName = array[0..79] of Char;
const
  FileNameBuf: TFileName = 'TEST.PAS';
  FileNamePtr: PChar = FileNameBuf;
```

Indizieren von Zeichenzeigern

Genauso wie ein **array**[0..X] of Char kompatibel ist mit einem Zeiger auf ein Zeichen, kann auch ein Zeichenzeiger indiziert werden, als ob er ein Zeichenarray wäre.

```
var
  A: array[0..63] of Char;
  P: PChar;
  Ch: Char;
  .
  .
begin
  P := A;
  Ch := A [5]
  Ch := P [5]
end;
```

Die letzten beiden Zuweisungen weisen *Ch* den Wert des sechsten Zeichenelements von *A* zu.

Wenn Sie einen Zeichenzeiger indizieren, gibt der Index an, daß ein nicht negativer Offset dem Zeiger hinzugefügt wird, bevor dieser Zeiger dereferenziert wird. Somit ist *P[0]* äquivalent zu *P*^ und spezifiziert das Zeichen, auf das mit *P* gezeigt wird. *P[1]* gibt das Zeichen rechts nach dem Zeichen an, auf das mit *P* gezeigt wird. *P[2]* gibt das nächste Zeichen an usw. Beim Indizieren verhält sich ein *PChar*-Typ so, als ob er definiert wäre wie

```
type
  TCharArray = array[0..65535] of Char;
  PChar = ^TCharArray;
```

Der Compiler führt bei der Indizierung eines Zeichenzeigers keine Bereichsprüfungen durch, da ihm keine Typinformationen zu Verfügung stehen, um die maximale Länge des null-terminierten Strings festzustellen, auf den der Zeichenzeiger zeigt. Solche Bereichsprüfungen müssen von Ihrem Programm durchgeführt werden.

Die folgende Funktion *StrUpper* zeigt, wie mittels Indizierung eines Zeichenzeigers ein null-terminierter String in Großbuchstaben umgewandelt wird.

```
function StrUpper(Str: Pchar): PChar
var
  I: Word;
begin
  I := 0;
  while Str[I] <> #0 do
  begin
    Str[I] := UpCase(Str[I]);
    Inc(I);
  end;
  StrUpper := Str;
end;
```

Beachten Sie, daß *StrUpper* eine Funktion und keine Prozedur ist, und daß sie immer den Wert liefert, der ihr als Parameter übergeben wurde. Da es die erweiterte Syntax ermöglicht, daß das Ergebnis eines Funktionsaufrufs ignoriert wird, kann *StrUpper* so behandelt werden, als ob es eine Prozedur sei:

```
StrUpper(A);
PrintStr(A);
```

Da *StrUpper* aber wie schon erwähnt immer den an sie übergebenen Wert zurückliefert, können die beiden obenstehende Anweisungen in einer Anweisung zusammengefaßt werden:

```
PrintStr(StrUpper(A));
```

Operationen mit Zeigern

Die erweiterte Syntax in Turbo Pascal gestattet einige neue Operationen mit Zeichenzeigern. Die Operatoren Plus (+) und Minus (-) können verwendet werden, um den Offset-Wert eines Zeigers zu erhöhen oder zu verringern. Mit dem Operator Minus kann der Abstand (die Differenz) zwischen den Offset-Werten zweier Zeichenzeiger berechnet werden. Vorausgesetzt, daß *P* und *Q* Werte des Typs *PChar* sind und *I* ein Wert des Typs *Word*, sind die folgenden Konstrukte zulässig:

*P+I*Addiere *I* zum Offset von *P*
*I+P*Addiere *I* zum Offset von *P*
*P+I*Addiere *I* zum Offset von *P*
*P-I*Subtrahiere *I* vom Offset von *P*
*P-Q*Subtrahiere den Offset von *Q* vom Offset von *P*

Die Operationen *P+I* und *I+P* addieren *I* zu der von *P* repräsentierten Adresse. Damit wird ein Zeiger erzeugt, der auf *I* Zeichen nach *P* zeigt. Die Operation *P-I* subtrahiert *I* von der von *P* übergebenen Adresse. Damit wird ein Zeiger erzeugt, der auf *I* Zeichen vor *P* zeigt.

Die Operation *P-Q* berechnet den Abstand zwischen *Q* (der niedrigeren Adresse) und *P* (der höheren Adresse). Dabei wird ein Wert vom Typ *Word* erzeugt, der die Anzahl der Zeichen zwischen *Q* und *P* angibt. Diese Operation

setzt voraus, das *P* und *Q in das selbe Zeichenarray zeigen*. Wenn die beiden Zeiger in verschiedene Zeichenarrays zeigen, ist das Ergebnis nicht definiert.

Die Standard-Syntax von Turbo Pascal ermöglicht nur, Zeiger zu vergleichen, um festzustellen ob sie gleich sind oder nicht. Die erweiterte Syntax ermöglicht weiterhin die Anwendung der Operatoren >, <, >= und <= auf Werte des Typs *PChar*. Beachten Sie aber, daß diese relationalen Prüfungen voraussetzen, daß die beiden zu vergleichenden Zeiger in dasselbe Zeichenarry zeigen, und daß aus diesem Grund die Operatoren nur die Offset-Werte der beiden Zeiger vergleichen. Wenn die Zeiger in verschiedene Zeichenarrays zeigen, ist das Ergebnis nicht definiert.

```
var
  A, B: array[0..79] of Char;
  P, Q: PChar;
begin
  P := A;              {P zeigt auf A [0]}
  Q := A+5;            {Q zeigt auf A [5]}
  if P < Q then ...;   {Gültiger Test, Ergebnis ist True}
  Q := B;              {Q zeigt auf B [10]}
  if P < Q then ...;   {Ergebnis ist nicht definiert}
end;
```

Null-terminierte Strings und Standard-Prozeduren

Turbo Pascals erweiterte Syntax ermöglicht, daß die Standardprozeduren *Read, Readln* und *Val* mit Arrays des Typs **array**[0..X] *of* Char verwendet werden können. Die Standard-Prozeduren *Write, Writeln, Val*, *Assign* und *Rename* können sowohl mit solchen Zeichenarrays als auch mit Zeigern auf Zeichen verwendet werden. Weitere Informationen zu den Standardprozeduren finden Sie in Kapitel 24 »Verzeichnis der Routinen«.

Was die Unit Strings definiert

Turbo Pascal für Windows hat keine eingebauten Routinen zur Handhabung von null-terminierten Strings. Stattdessen finden Sie diese Funktionen in der Unit *Strings*. Im folgenden listen wir diese Funktionen auf, versehen mit einer kurzen Beschreibung.

Tabelle 13.1 Funktionen in der Unit Strings

Funktion	Beschreibung
StrLen	liefert die Länge eines Strings.
StrEnd	liefert einen Zeiger auf das Ende eines Strings, also einen Zeiger auf das Null-Zeichen, das den Strings abschließt.
StrMove	kopiert eine anzugebende Anzahl Zeichen des Quellstrings in den Zielstring. Quell- und Zielbereich können sich überschneiden.
StrCopy	kopiert den Quellstring in einen Zielstring und liefert einen Zeiger auf den Zielstring.
StrECopy	kopiert den Quellstring in einen Zielstring und liefert einen Zeiger auf das Ende des Zielstrings.
StrLCopy	kopiert eine vorgegebene Anzahl von Zeichen von einem Quellstring in einem Zielstring und liefert einen Zeiger auf den Zielstring.
StrPCopy	kopiert einen Pascal-String in einen null-terminierten String und liefert einen Zeiger auf den null-terminierten String.
StrCat	hängt einen Quellstring an das Ende eines Zielstrings und liefert einen Zeiger auf den Zielstring.
StrLCat	hängt einen Quellstring an das Ende eines Zielstrings und überprüft dabei, daß die Länge des sich daraus ergebenden Strings nicht ein vorgegebene Maximum überschreitet. Zurückgeliefert wird ein Zeiger auf den Zielstring.
StrComp	vergleicht zwei Strings, S1 und S2, und liefert einen Wert kleiner Null wenn S1 > S2, den Wert Null wenn S1 = S2 und einen Wert größer Null, wenn S1 > S2.
StrIComp	vergleicht zwei Strings ohne Berücksichtigung von Groß- und Kleinschreibung.
StrLComp	vergleicht zwei Strings über eine vorgegebene maximale Länge.
StrLIComp	vergleicht zwei Strings über eine vorgegebene maximale Länge ohne Berücksichtigung von Groß- und Kleinschreibung.
StrPas	konvertiert einen null-terminierten String in einen Pascal-String.
StrPos	liefert einen Zeiger auf das erste Vorkommen einer bestimmten Zeichenfolge in einem String oder **nil, wenn diese Zeichenfolge im String nicht vorkommt.**
StrRScan	liefert einen Zeiger zum letzten Vorkommen eines vorgegebenen Zeichens in einem String, oder **nil**, wenn das Zeichen nicht im String vorkommt.
StrScan	liefert einen Zeiger zum ersten Vorkommen eines vorgegebenen Zeichens in einem String, oder **nil**, wenn das Zeichen nicht im String vorkommt.
StrUpper	konvertiert einen String zu Großbuchstaben und liefert einen Zeiger auf den String.
StrLower	konvertiert einen String zu Kleinbuchstaben und liefert einen Zeiger auf den String.
StrNew	reserviert einen String auf dem Heap.
StrDispose	gibt den Speicherplatz, den ein String belegt hat, wieder frei.

Das folgenden Beispiel zeigt, wie die Funktion *FileSplit* in der Unit *WinDos* einige dieser String-Funktionen einsetzt.

```
{ Maximale Länge der Dateinamenskomponenten }

const
  fsPathName = 79;
  fsDirectory = 67;
  fsFileName = 8;
  fsExtension = 4;'

{ FileSplit Rücksprungs-Flag }

const
  fcExtension = $0001;
  fcFileName = $0002;
  fcDirectory = $0004;
  fcWildcards = $0008;

{ FileSplit teilt Dateinamen, die mit Path angegeben werden, in seine drei Komponenten. Dir speichert den Laufwerksbezeichner und den Verzeichnispfad mit vorangestelltem und folgendem Backslash, Name den Dateinamen und Ext die Dateinamenserweiterung, der ein Punkt vorangestellt wird. Ist einer dieser String-Parameter NIL, wird die entsprechende Komponente nicht gespeichert. Wenn der in Path übergebenene Pfad eine der Komponenten nicht enthält, ist der gelieferte Komponentenstring leer. Die maximale Länge der Strings, die in Dir, Name und Ext geliefert werden, werden von den Konstanten fsDirectory, fsFileName und fsExtension definiert. Der zurückgelieferte Wert ist eine Kombination aus den Bitmasken von fcDirectory, fcFileName und fcExtension, und gibt an, welche Komponenten im Pfad vorhanden waren. Wenn ein Name oder eine Dateinamenerweiterung eine Jokerzeichen (* oder ?) enthält, wird das Flag fcWildcard im gelieferten Wert gesetzt. }

function Filesplit (Path, Dir, Name, Ext: PChar): Word;
var
  DirLen, NameLen, Flags: Word;
  NamePtr, ExtPtr: PChar;
begin
  NamePtr := StrRScan (Path, '\');
  if NamePtr = nil then NamePtr := StrScan (Path,':');
  if NamePtr = nil then NamePtr := Path else Inc(NamePtr);
  ExtPtr := SrScan (NamePtr,'.');
  if ExtPtr =  nil then ExPtr := StrEnd(NamePtr);
  DirLen := NamePtr - Path;
  if DirLen > fsDirectory then DirLen := fsDirectory;
  NameLen := ExtPtr - NamePtr;
  if NameLen > fsFilename then NameLen .=fsFilename;
  Flags := 0;
  if (StrScan(NamePtr,'?') <> nil) or
```

```
      (StrScan(NamePtr,'*') <> nil) then
      Flags := fcWildcards;
    if DirLen <> 0 then Flags := Flags or fcDirectory;
    if NameLen <> 0 then Flags := Flags or fcFilename;
    if ExtPtr [0] <> #0 then Flags := Flags or fcExtension;
    if Dir <> nil then StrLCopy(Dir, Path, DirLen);
    if Name <> nil then StrLCopy(Name, NamePtr, NameLen);
    if Ext <> nil then StrLCopy(Ext, ExtPtr, fsExtension);
    FileSplit := Flags;
  end;
```

14. Die Unit WinCrt

Die Unit *WinCrt* implementiert ein terminal-ähnliches Textfenster unter Windows. Mit *WinCrt* können Sie genauso unkompliziert wie im Textmodus ein Programm schreiben, das die Standardprozeduren *Read, Readln, Write* und *Writeln* verwendet, um Eingabe- und Ausgabeoperationen durchzuführen. *WinCrt* enthält alle logischen Steuerfunktionen, die benötigt werden, um ein Textfenster unter Windows zu emulieren. Wenn Ihr Programm also *WinCrt* verwendet, müssen Sie keinen »Windows-spezifischen« Programmcode erstellen.

Einbindung der Unit WinCrt

Um die Unit *WinCrt* in Ihrem Programm einzubinden, tragen Sie sie in der uses-Anweisung Ihres Programms ein.

Als Voreinstellung sind die in der Unit *Systems* definierten Standard-Textdateien *Input* und *Output* nicht zugewiesen. Alle *Read-, Readln-, Write-* und *Writeln*-Prozeduraufrufe ohne Angabe einer Dateivariablen führen daher zu einem I/O-Fehler. Wenn ein Programm aber die Unit *WinCrt* verwendet, weist der Initialisierungscode der Unit die Textdateien *Input* und *Output* so zu, daß sie sich auf ein Fenster beziehen, das ein Textfenster emuliert. Der Effekt ist derselbe als ob die folgenden Anweisungen am Anfang des Programms angegeben worden wären:

```
AssignCrt(Input); Reset(Input);
AssignCrt(Output); Rewrite(Output);
```

Wenn der erste *Read-, Readln-, Write-* oder *Writeln*-Aufruf in Ihrem Programm ausgeführt wird, öffnet sich ein Fenster auf dem Windows-Desktop. Der voreingestellte Name dieses Fensters ist der volle Pfadname der .EXE-Datei des Programms. Wenn ein Programm beendet wird, (wenn das reservierte Wort end erreicht wird), ändert sich der Titel des CRT-Fenster zu »*(Inaktiv nnnnn)*«, wobei *nnnnn* der Titel des Fensters im aktiven Zustand ist.

Beachten Sie, daß das Fenster geöffnet bleibt, obwohl das Programm beendet ist. Dadurch können Sie die Bildschirmausgaben des Programms überprüfen.

Entsprechend anderen Windows-Programmen wird das Programm nicht vollständig beendet, solange der Benutzer das Fenster nicht schließt.

Die Routinen *InitWinCrt* und *DoneWinCrt* erlauben Ihnen eine genauere Kontrolle über eines Textfensters. Ein Aufruf an *InitWinCrt* erzeugt sofort ein solches Fenster, anstatt auf den ersten Aufruf von *Read, Readln, Write* oder *Writeln* zu warten. Entsprechend schließt der Aufruf von *DoneWinCrt* sofort das Fenster, anstatt dies dem Benutzer zu überlassen.

Ein Textfenster ist ein verschiebbarer Ausschnitt auf einem virtuellen Textbildschirm. Voreingestellt ist eine Größe von 80 Spalten und 25 Zeilen, tatsächlich kann das Fenster aber kleiner sein. Ist das der Fall, können Sie die Bildlaufleisten oder die Cursor-Tasten verwenden, um das Fenster innerhalb des größeren Textbildschirm zu bewegen. Die ist besonders beim »Zurückrollen« zur Überprüfung von vorher geschriebenem Text nützlich. In der Voreinstellung folgt das Ausschnittsfenster dem Cursor im Textbildschirm. Mit anderen Worten, das Ausschnittsfenster verschiebt automatisch den dargestellten Text, um sicherzustellen, daß der Cursor immer sichtbar ist. Sie können diese automatische Tracking-Funktion ausschalten, indem Sie die Variable *AutoTracking* auf *False* setzen.

Die Maße eines virtuellen Textbildschirms werden über die Variable *ScreenSize* festgelegt. Sie können diese Maße ändern, indem Sie *ScreenSize* neue Werte zuweisen, bevor das Programm ein Textfenster öffnet. Bei der Erzeugung eines Fensters wird ein Bildschirmpuffer im dynamischen Speicher angelegt. Die Größe dieses Puffers wird durch *ScreenSize.X* multipliziert mit *ScreenSize.Y* berechnet, er kann jedoch nicht größer als 65520 Bytes sein. Sie müssen also sicherstellen, daß durch die Werte, die Sie für *ScreenSize.X* und *ScreenSize.Y* angeben, dieser Wert nicht überschritten wird. Wenn Sie zum Beispiel *ScreenSize.Y* mit dem Wert 64 belegen, darf für *ScreenSize.Y* höchstens der Wert 1023 angegeben werden.

Während ein Programm, das die Unit *WinCrt* verwendet, abläuft, kann der Benutzer jederzeit dieses Programm beenden, indem er den Befehl *Schließen* im Systemmenü des Textfensters anwählt, zweimal auf das Menüsymbol klickt oder [Alt][F4] drückt. Außerdem kann durch Drücken von [Ctrl][C] oder [Ctrl][Break] eine Anwendung jederzeit angehalten werden und das Fenster in den inaktiven Zustand versetzt werden. Diese Funktionen können ausgeschaltet werden, indem die Variable *CheckBreak* zu Beginn des Programms auf *False* gesetzt wird.

Sonderzeichen

Bei Schreibaktionen über *Output* oder in eine Datei, der einem *CRT*-Fenster zugeordnet worden ist, haben die folgenden Zeichen eine besondere Bedeutung:

Zeichen	Wirkung
#7 (BEL)	erzeugt einen Piepton im eingebauten Lautsprecher Ihres Rechners
#8 (BS)	bewegt den Cursor um eine Position nach links. Wenn er sich bereits am linken Fensterrand befindet, passiert gar nichts
#10 (LF)	bewegt den Cursor um eine Zeile nach unten, ohne seine Spaltenposition zu verändern. Wenn sich der Cursor bereits in der untersten Zeile des Fensters befindet, wird der gesamte Fensterinhalt entsprechend nach oben gerollt
#13 (CR)	setzt den Cursor auf den linken Fensterrand, ohne seine vertikale Position zu verändern

Alle anderen Zeichen werden ohne Interpretation ausgegeben.

Zeilen-Input

Leseaktionen von der Standard-Datei *Input* oder einer Textdatei, der ein Textfenster zugeordnet worden ist, geschehen zeilenweise: eine gelesene Eingabezeile wird im internen Puffer der jeweiligen Datei-Variablen zwischengespeichert und von dort aus je nach Bedarf gelesen. Erst wenn der Inhalt dieses Puffers »ausgelesen« ist und das Programm weitere Eingaben erwartet, folgt eine weitere physikalische Eingabeanforderung.

Bei der zeilenweisen Eingabe in ein Textfenster kann mit Backspace das zuletzt eingegebene Zeichen gelöscht werden. Mit Enter wird die Eingabezeile abgebrochen und eine Zeilenendemarkierung (CR/LF) im Buffer gespeichert. Auch [Ctrl][Z] bricht die Eingabezeile ab und erzeugt eine Dateiende-Markierung, wenn die Variable *CheckEOF* auf *True* gesetzt ist (ihre Voreinstellung ist *False*).

Mit den Funktionen *KeyPressed* und *ReadKey* kann der Tastaturstatus überprüft und einzelne Zeichen unter Programmkontrolle eingegeben werden.

Variablen

Die Unit *WinCrt* deklariert mehrere typisierte Konstanten und eine Variable. Die typisierten Konstanten sind zwar auf bestimmte Anfangswerte gesetzt, können jedoch als Variablen angesehen werden, da Sie diese Anfangswerte Ihren Wünschen entsprechend ändern können.

Variable	Typ
WindowOrg	TPoint
WindowSize	TPoint
ScreenSize	TPoint
Cursor	TPoint
Origin	TPoint
InactiveTitle	TPoint
AutoTracking	Boolean
CheckEOF	Boolean
CheckBreak	Boolean
WindowTitle	array [0..79] of Char

WindowOrg
Legt die Anfangsposition des Textfensters fest.

```
const WindowOrg: TPoint = (X: cw_UseDefault; Y: cw_UseDefault);
```

Über die voreingestellte Position kann Windows eine geeignete Position für das Textfenster wählen. Die Anfangsposition kann geändert werden, indem den X- und Y-Koordinaten neue Werte zugewiesen werden, bevor das Fenster geöffnet wird.

WindowSize
Legt die Anfangsgröße des Textfensters fest.

```
const WindowSize: TPoint = (X: cw_UseDefault; Y: cw_UseDefault);
```

Über die voreingestellten Maße kann Windows die geeignete Größe für das Textfenster festlegen. Die Anfangsgröße kann geändert werden, indem den X- und Y-Koordinaten neue Werte zugewiesen werden, bevor das Textfenster geöffnet wird.

ScreenSize
Legt die Breite und Höhe des virtuellen Bildschirms im Textfenster in Zeichen fest.

```
const ScreenSize: TPoint = (X: 80; Y: 25);
```

Die voreingestellte Bildschirmgröße ist 80 Spalten mal 25 Zeilen. Sie können die Größe des virtuellen Bildschirms ändern, indem Sie den X- und Y-Koordinaten von *ScreenSize* andere Werte zuweisen, bevor ein Textfenster erzeugt wird.
Der Wert, der von *ScreenSize.X* mal *ScreenSize.Y* angegeben wird, darf nicht größer als 65520 sein.

Cursor
Speichert die aktuelle Position des Cursors im virtuellen Bildschirm.

```
const Cursor: TPoint = (X: 0; Y: 0);
```

Die obere linke Ecke entspricht (0,0). *Cursor* ist eine schreibgeschützte Variable, weisen Sie ihr also keine Werte zu.

Origin
Enthält die virtuellen Bildschirmkoordinaten des Zeichenzelle, die in der oberen linken Ecke des CRT-Fensters angezeigt wird.

```
const Origin: TPoint = (X: 0; Y: 0);
```

Origin ist eine schreibgeschützte Variable, weisen Sie ihr also keine Werte zu.

InactiveTitle
Zeigt auf einen null-terminierten String, der verwendet wird, wenn ein Titel in einem nicht-aktiven Textfenster konstruiert wird.

```
const InactiveTitle: PChar = '(Inactive %s);
```

Der String wird als Format-Kontrollparameter eines Aufrufs an die Windows-Funktion *WVSPrintF* verwendet. Der Platzhalter *%s* gibt, falls vorhanden, an, wo der Fenstertitel eingefügt werden soll.

AutoTracking
Aktiviert bzw. deaktiviert das automatische Verschieben des Fensterinhalts, damit der Cursor sichtbar bleibt.

```
const AutoTracking: Boolean = True;
```

Wenn *AutoTracking* auf *True* gesetzt ist, wird der Fensterinhalt automatisch verschoben, um sicherzustellen, daß der Cursor nach jeder *Write*- und *Writeln*-Operation sichtbar ist. Der Text, der in das Fenster geschrieben wird, wird nicht automatisch angezeigt.

CheckEOF
Aktiviert und deaktiviert des Zeilenende-Zeichen.

```
const CheckEOF : Boolean = False;
```

Wenn *CheckEOF* auf *True* gesetzt ist, wird eine Zeilenende-Markierung erzeugt, wenn der Benutzer [Ctrl][Z] drückt, während aus einer Datei gelesen wird, der das Textfenster zugewiesen ist. Wenn *CheckEOF* auf *False* gesetzt ist, hat das Drücken von [Ctrl][Z] keine Auswirkungen.

CheckBreak
Aktiviert und deaktiviert die Möglichkeit, daß der Benutzer die Anwendung abbricht.

```
const CheckBreak: Boolean = True;
```

Wenn *CheckBreak* auf *True* gesetzt ist, kann der Benutzer jederzeit das Programm beenden, indem er den Befehl *Schließen* im Systemmenü des Textfensters anwählt, zweimal auf das Menüsymbol klickt oder [Alt][F4] drückt. Außerdem kann durch Drücken von [Ctrl][C] oder [Ctrl][Break] eine Anwendung jederzeit angehalten werden und das Fenster in den inaktiven Zustand versetzt werden. Diese Funktionen können ausgeschaltet werden, indem die Variable *CheckBreak* zu Beginn des Programms auf *False* gesetzt wird.

WindowTitle
Legtt den Titel des Textfensters fest.

```
var WindowTitle: array[0..79] of Char
```

In der Voreinstellung wird der gesamte Pfadname der .EXE-Datei des Programms als Fenstertitel verwendet. Sie können diesen Titel ändern, indem Sie *WindowTitle* einen neuen String zuweisen, bevor das Textfenster erzeugt wird. Ein Beispiel:

```
StrCopy (WindowTitle, 'Hallo Erdling');
```

Prozeduren und Funktionen

In der folgenden Tabelle finden Sie die Prozeduren und Funktion, die in der Unit *WinCrt* definiert sind.

Prozedur	Beschreibung
AssignCrt	ordnet den Textbildschirm einer Textdatei zu
ClrEol	löscht sämtliche Zeichen ab der Position des Cursors bis zum Zeilenende
ClrScr	löscht den gesamten Fensterinhalt und setzt den Cursor in die obere linke Ecke des Fensters
CursorTo	bewegt den Cursor an die angegebenen Koordinaten im virtuellen Bildschirm
DoneWinCrt	entfernt das Textfenster
GotoXY	setzt den Cursor auf die angegebene Spalten- und Zeilenposition im virtuellen Fenster
InitWinCrt	erzeugt ein Textfenster.
ScrollTo	verschiebt den Fensterinhalt, um die Bildschirmposition darzustellen.
TrackCursor	verschiebt den Fensterinhalt, um sicherzustellen, daß der Cursor sichtbar bleibt
WriteBuf	schreibt einen Block von Zeichen in das Textfenster
WriteChar	schreibt ein einzelnes Zeichen in das Textfenster

Funktion	Beschreibung
KeyPressed	prüft die Tastatur und liefert *True*, wenn eine Taste gedrückt wurde
ReadBuf	ermöglicht die Eingabe einer Textzeile innerhalb des Fensters
ReadKey	liest ein Zeichen von der Tastatur
WhereX	liefert die X-Koordinate der aktuellen Cursorposition.
WhereY	liefert die Y-Koordinate der aktuellen Cursorposition.

Im folgenden Abschnitt werden die Prozeduren und Funktionen ausführlicher beschrieben.

InitWinCrt
Erzeugt ein Textfenster, sofern noch keines geöffnet ist.

```
procedure InitWinCrt;
```

Die Standardprozeduren *Read, Readln, Write* und *Writeln* rufen automatisch *InitWinCrt* auf, um sicherzustellen, daß das Textfenster existiert. *InitCrt* verwendet die Konstanten *WindowOrg*, *WindowSize* und *ScreenSize* und die Variable *WindowTitle*, um die Charakteristika des Textfensters festzulegen.

DoneWinCrt
Entfernt das Textfenster, falls dies noch nicht geschehen ist.

```
procedure DoneWinCrt
```

Der Aufruf von *DoneWinCrt* kurz vor Programmende verhindert, daß das Textfenster in den inaktiven Zustand versetzt wird. Der Benutzer muß das Fenster nicht manuell schließen.

WriteBuf
Schreibt einen Block von Zeichen in das Textfenster.

```
procedure WriteBuf(Buffer: PChar; Count: Word);
```

Buffer zeigt auf das erste Zeichen des Blocks. *Count* enthält die Anzahl der Zeichen, die geschrieben werden. Wenn *AutoTracking* auf *True* gesetzt ist, wird, wenn notwendig, der Fensterinhalt verschoben, um sicherzustellen, daß der Cursor angezeigt wird, nachdem der Zeichen-Block in das Fenster geschrieben wurde.

WriteChar
Schreibt ein einzelnes Zeichen in das Textfenster.

```
procedure WriteChar (Ch: Char);
```

KeyPressed
Liefert *True*, wenn eine Taste auf der Tastatur gedrückt wurde, andernfalls *False*.

```
function KeyPressed: Boolean;
```

Die Taste kann mit der Funktion *ReadKey* gelesen werden.

ReadKey
Liest eine Zeichen von der Tastatur.

```
function ReadKey; Char;
```

Das Zeichen wird nicht auf dem Bildschirm angezeigt.

Hinweis: Die Funktion *ReadKey* unterstützt nur Standard-ASCII-Codes. Erweiterte Tastaturcodes, zum Beispiel die Funktions- und Cursortasten-Codes, werden von *ReadKey* nicht unterstützt.

ReadBuf
Liest eine Zeile aus dem Textfenster ein.

```
function ReadBuf(Buffer: PChar; Count: Word): Word;
```

Buffer zeigt auf einen Zeilenpuffer, der für bis zu der mit *Count* angegebenen Anzahl von Zeichen Platz hat. Es können bis zu *Count - 2* Zeichen eingegeben werden. Eine Zeilenende-Markierung (#13 gefolgt von #10) wird automatisch an die Zeile angehängt, wenn der Benutzer [Enter] drückt. Wenn *CheckEOF* auf *True* gesetzt ist, kann der Benutzer die Eingabezeile auch mit [Ctrl][Z] abbrechen. In diesem Fall wird der Zeile eine Dateiende-Markierung (#26) angehängt. Die Funktion liefert die Anzahl der gelesenen Zeichen, inklusive Zeilenende- und Dateiende-Markierung zurück.

GotoXY
Setzt den Cursor auf die angegebene Spalten- und Zeilenposition im virtuellen Fenster.

```
procedure GotoXY (X, Y:Integer);
```

Die obere linke Ecke entspricht (1,1). Die Variable *Cursor* ist auf (X-1, Y-1) gesetzt, da sie die Cursorposition relativ zu (0,0) anstatt zu (1,1) speichert.

Hinweis: Die Routinen *GotoXY, WhereX* und *WhereY* sind in erster Linie zur Kompatibilität mit der Unit *Crt* aus der DOS-Version von Turbo Pascal gedacht. Die auf den Ursprung (1,1) bezogenen Koordinaten dieser Routinen sind nicht konsistent mit den übrigen Routinen und Variablen in der Unit *WinCrt*. Wir empfehlen deshalb die Verwendung der Variablen *CursorTo* und *Cursor*.

WhereX
Liefert die X-Koordinate der aktuellen Cursorposition.

```
function WhereX: Integer;
```

Der gelieferte Wert basiert auf dem Ursprung (1,1) und entspricht dem Wert *Cursor.X + 1*.

WhereY
Liefert die Y-Koordinate der aktuellen Cursorposition.

```
function WhereY: Integer;
```

Der gelieferte Wert basiert auf dem Ursprung (1,1) und entspricht dem Wert *Cursor.Y + 1*.

ClrScr
Löscht den gesamten Fensterinhalt und setzt den Cursor in die obere linke Ecke des Fensters.

```
procedure ClrScr;
```

ClrEol
Löscht sämtliche Zeichen ab der Position des Cursors bis zum Zeilenende.

```
procedure ClrEol;
```

CursorTo
Bewegt den Cursor an die angegebenen Koordinaten im virtuellen Bildschirm.

```
procedure CursorTo (X, Y: Integer);
```

Die obere linke Ecke entspricht den Koordinaten (0,0). Die Variable *Cursor* wird auf (X,Y) gesetzt.

ScrollTo
Verschiebt den Inhalt des Textfensters, um den über (X,Y) angegebenen Bereich des virtuellen Textfensters an der oberen linken Ecke darzustellen.

```
procedure ScrollTo(X, Y: Integer);
```

(0,0) entspricht der oberen linken Ecke des virtuellen Bildschirms. Die Variable *Origin* wird auf (X,Y) gesetzt.

TrackCursor
Verschiebt falls nötig den Inhalt des Textfensters, um sicherzustellen, daß der Cursor sichtbar bleibt.

```
procedure TrackCursor
```

AssignCrt
Verbindet eine Textdatei mit dem Textfenster.

```
procedure AssignCrt (var F: Text);
```

Folgende *Write*- und *Writeln*-Operationen in die Datei *F* schreiben in das Textfenster. *Read*- und *Readln*-Operationen lesen aus dem Textfenster.

15. Der Coprozessor 8087

Turbo Pascal unterscheidet wie die meisten anderen Programmiersprachen zwischen zwei Arten numerischer Werte: Integer (Typen *ShortInt*, *Integer*, *LongInt*, *Byte*, *Word*) und Realzahlen (Typen *Real*, *Single*, *Double*, *Extended*, *Comp*), die aufgrund des internen Speicherformats auch als Gleitkommazahlen bezeichnet werden.

Der Prozessor 8086 verfügt über eine ganze Reihe von Befehlen für arithmetische Operationen mit Integern, die Verarbeitung von Realzahlen muß dagegen über (aufwendige und zeitraubende) Unterprogramme erfolgen. Aus diesem Grund wurde der 8087 entworfen – ein spezieller Chip, der Berechnungen mit Fließkommazahlen über direkte Befehle (und deshalb mit hoher Geschwindigkeit) ausführen kann. Wenn Sie viel mit derartigen Werten arbeiten, sollten Sie sich die Anschaffung dieses Chips überlegen – die Ausführungsgeschwindigkeit eines Programms läßt sich durch einen 8087 erheblich steigern.

Turbo Pascal unterstützt sowohl Systeme mit als auch ohne installiertem Coprozessor:

- Für Programme, die hauptsächlich auf PCs ohne Coprozessor laufen sollen, bietet die Laufzeitbibliothek den Datentyp *Real* und eine dazugehörige Sammlung von Unterprogrammen, in denen Fließkomma-Operationen durch den 8086 ausgeführt werden. Eine Variable vom Typ *Real* belegt sechs Bytes Speicherplatz, hat einen zulässigen Wertebereich von $2.9*10^{-39}$ bis $1.7*10^{38}$ und eine Genauigkeit von 11 bis 12 Dezimalstellen. Die Bibliotheksroutinen sind vor allem hinsichtlich der Geschwindigkeit und des Code-Umfangs optimiert – deshalb fehlen einige spezielle Funktionen, deren Programmierung ohne Coprozessor reichlich aufwendig wäre.

- Für Programme, bei denen es auf hohe Rechengenauigkeit ankommt, kann der Compiler Code erzeugen, der den 8087 direkt ansteuert. Damit stehen Ihnen vier weitere Datentypen (*Single*, *Double*, *Extended* und *Comp*) zur Verfügung, der zulässige Wertebereich geht von $1.9*10^{-4951}$ bis $1.1*10^{4932}$ mit einer Genauigkeit von 19-20 Dezimalstellen.

Über den Compiler-Befehl `$N` bzw. innerhalb der Entwicklungsumgebung im Dialogfenster *Options/Compiler* kann zwischen den beiden Modi der Code-Erzeugung umgeschaltet werden. `$N-` steht für die softwaremäßige Behandlung von Gleitkomma-Operationen und ist die Standardvorgabe, `$N+` erzeugt Code

zur Ansteuerung des Coprozessors. In diesem Modus stehen höhere Genauigkeit und vier weitere Real-Datentypen zur Verfügung.

Sie müssen sicherstellen, daß Windows auf die Bibliothek WIN87EM.DLL zugreifen kann, wenn Sie ein Programm, das im Modus {$N+} kompiliert wurde, starten wollen. Diese zu Windows gehörende Datei enthält die Emulationsroutinen für den 8087 und stellt die Schnittstelle dar zwischen Ihrem Programm, Windows und dem Coprozessor 80X87. Ist dieser Prozessor in Ihrem System nicht installiert, *emulieren* Routinen aus dieser Bibliothek den 80X87. Eine solche Emulation ist natürlich langsamer als ein Coprozessor, aber immerhin ist damit sichergestellt, daß ein Programm, das mit dem Coprozessor arbeitet, auch dann läuft, wenn kein 80X87 vorhanden ist.

Wenn Sie ein Programm, das im Modus {$N+} kompiliert wurde, starten, muß die Bibliothek WIN87EM.DLL auch dann für Windows zur Verfügung stehen, wenn Ihr Computer mit einem 80X87 ausgerüstet ist.

Hinweis: Die mathematischen Routinen in *System* (*Pi*, *Sqrt*, *Sin* usw.) liefern Werte vom Typ *Real* zurück, wenn ohne Coprozessor gearbeitet wird ({$N-}). Im Modus $N+ haben die Ergebnisse dagegen den Typ *Extended*:

```
{$N+}
begin
  Writeln(Pi);  { Ergebnis 3.14159265358979E+0000 }
end.
```

```
{$N-}
begin
  Writeln(Pi);  { Ergebnis 3.1415926536E+00 }
end.
```

Die folgenden Abschnitte dieses Kapitels erläutern die Zusammenarbeit zwischen Turbo Pascal und einem numerischen Coprozessor im Detail.

Die Datentypen des 8087

Im Modus {$N+} stellt Turbo Pascal vier weitere Datentypen für numerische Werte zur Verfügung:

- Der Typ *Single* ist das »kleinste« Format, in dem sich eine Realzahl speichern läßt: jede Variable dieses Typs belegt vier Bytes Speicherplatz, hat einen Wertebereich von $1.5*10^{-45}$ bis $3.4*10^{38}$ und eine Genauigkeit von 7 bis 8 Dezimalstellen.

- Der Typ *Double* belegt genau doppelt soviel Speicherplatz wie der Typ *Single*, nämlich 8 Bytes pro Variable. Sein Wertebereich geht von $5.0*10^{-324}$ bis $1.7*10^{308}$, die Genauigkeit liegt zwischen 15 und 16 Dezimalstellen.

- Der Typ *Extended* stellt das »größte« Format dar, das vom 8087 direkt unterstützt wird: jede Variable dieses Typs belegt 10 Bytes Speicherplatz. Ent-

sprechend gewaltig ist der zulässige Wertebereich, der sich von $1.9*10^{4951}$ bis $1.1*10^{4932}$ erstreckt.

- Der Typ *Comp* speichert ganzzahlige Werte im Bereich von -2^{63} bis $2^{63}-1$ (ca. $-9.2*10^{18}$ bis $9.2*10^{18}$). Eine Variable dieses Typs belegt 8 Bytes Speicherplatz und ist eigentlich eine Art doppeltgenauer LongInt – da Comp-Werte aber durch den Coprozessor bearbeitet werden, betrachtet man sie im allgemeinen als Realzahlen. Der Typ Comp ist für Berechnungen aus dem Finanzbereich hervorragend geeignet, weil Operationen mit ihm keine Rundungsfehler erzeugen, wenn beispielsweise mit der Grundeinheit »Pfennig« gearbeitet wird.

Der Typ *Real* ist immer verfügbar - unabhängig davon, ob Sie im Modus {$N+} oder {$N-} arbeiten bzw. ob Ihr Computer mit einem Coprozessor ausgerüstet ist oder nicht: Programme, die ohne Unterstützung durch einen 8087 geschrieben wurden, müssen auch für den Betrieb mit einem 8087 nicht modifiziert werden, mit *Real* erzeugte Dateien sind zwischen beiden Code-Modellen uneingeschränkt austauschbar.

Ein Problem soll hier allerdings nicht verschwiegen werden: Im Modus {$N+} werden Operationen mit *Real*-Variablen langsamer ausgeführt – selbst dann, wenn das System mit einem 8087 ausgerüstet ist. Der Grund: Werte dieses Formats können vom Coprozessor nicht direkt verarbeitet werden, vor und nach jeder Berechnung ist deshalb eine (durch den 8086 ausgeführte und sehr aufwendige) Konvertierung notwendig. Da der Emulator den 8087 zu hundert Prozent simuliert, ist diese Konvertierung im Modus {$N+} auch auf Systemen ohne Coprozessor notwendig – hier sinkt die Ausführungsgeschwindigkeit erheblich.

Aus diesem Grund sollten Programme, die nicht ausschließlich für Systeme ohne 8087 geschrieben werden, mit den Datentypen *Single*, *Double*, *Extended* und *Comp* arbeiten.

Arithmetische Operationen im Format Extended

Der Typ *Extended* ist die Grundlage aller Berechnungen, die über den 8087 ausgeführt werden: er wird folglich von Turbo Pascal für die Speicherung von Konstanten und für sämtliche numerischen Ergebnisse verwendet, die keine ganzen Zahlen sind. Im folgenden Beispiel wird also die gesamte rechte Seite der Zuweisung – unabhängig von der Variablen auf der linken Seite – im Format *Extended* berechnet:

```
{$N+}
var
  X, A, B, C: Real;

begin
  X := (B + Sqrt(B * B - A * C)) / A;
end.
```

Diese Automatik hat gleich mehrere Vorteile: Rundungsfehler während der Berechnung können so gut wie vernachlässigt werden, Über- oder Unterschreitungen des Rechenbereichs sind so gut wie ausgeschlossen. Das Ergebnis sind stabilere und zuverlässigere Programme.

Die Eigenschaften des Typs *Extended* lassen sich auch für aufeinanderfolgende Berechnungen ausnutzen, bei denen Turbo Pascal nicht mehr automatisch für eine erhöhte Genauigkeit sorgen kann. Das folgende Beispiel berechnet die Normalbruchreihe, wobei das Ergebnis schließlich als einfachgenaue Realzahl gespeichert wird:

```
var
  Bruchsumme: Single;
  Nenner: Integer;
  T: Extended;   { zur Speicherung der Zwischenergebnisse }
begin
  T := 0.0;
  for Nenner := 1 to 20000 do
    T := T + 1.0 / Nenner;
  Bruchsumme := T;
end.
```

(Sie sollten dieses Programm einmal ausführen lassen und mit einer zweiten Version vergleichen, die *T* als *Single* deklariert. In diesem Fall summieren sich die Rundungsfehler bis auf die vierte Stelle hinter dem Komma auf).

Formale Parameter und Funktionsergebnisse können ebenfalls als *Extended* deklariert werden, um unnötige Konvertierungen und die dabei entstehenden Rundungsfehler zu vermeiden. Ein Beispiel:

```
function Flaeche(Radius: Extended): Extended;
begin
  Flaeche := Pi * Radius * Radius;
end;
```

Vergleiche von Realzahlen

Da Realzahlen grundsätzlich *Näherungswerte* darstellen, ergeben Vergleiche zwischen Variablen verschiedener Formate nicht immer die erwarteten Ergebnisse. Ein Beispiel:

```
var
  X: Single;
  Y: Double;

begin
  X := 1/3;  Y := 1/3;
  Writeln(X = Y);
end.
```

Das Ergebnis dieses Programms ist *False*, weil der in *X* gespeicherte Wert auf 7 bis 8 Dezimalstellen gerundet ist, der in *Y* gespeicherte Wert dagegen auf 15 bis 16 Stellen – nach der Konvertierung in das Format *Extended* unterscheiden sich beide etwa ab der 8. Stelle nach dem Komma. Aus demselben Grund ergeben die Befehle

```
X := 1/3;
Writeln(X = 1/3);
```

das Ergebnis *False*, wenn *X* als *Single* oder *Double* deklariert ist – Turbo Pascal berechnet den Ausdruck *1/3* mit einer Genauigkeit von 20 Dezimalstellen.

Der Rechenstack des 8087

Der 8087 verfügt über einen internen Stack, in dem bis zu 8 Werte gespeichert werden können. Zugriffe auf diesen Stack geschehen wesentlich schneller als das Auslesen entsprechender RAM-Zellen – folglich wird er von Turbo Pascal zur Speicherung von Operanden und Zwischenergebnissen verwendet.

Theoretisch ließe sich der Stack des 8087 mit sehr komplizierten Berechnungen zum Überlaufen bringen. Praktisch bildet die Begrenzung auf 8 Werte allerdings kein ernsthaftes Hindernis: Ausdrücke, die mehr als 8 Zwischenergebnisse erzeugen, dürften ausgesprochen selten vorkommen.

Wesentlich reeller sind die Gefahren, die in verschachtelten und/oder rekursiven Funktionsaufrufen liegen – falls derartige Konstrukte nicht solide durchdacht sind, kann der Stack des 8087 sehr wohl überladen werden. Ein Beispiel:

```
X := Test(A,B, Test(C,D, Test(E,F, Text(X,Y,Z))));
```

Hier enthält der Stack beim Aufruf von *Test(X,Y,Z)* bereits sechs Werte – nachdem noch drei weitere dazukommen, ist er überladen.

Korrekt programmiert lautet die Zuweisung:

```
X := Test(X,Y,Z);
X := Test(E,F,X);
X := Test(C,D,X);
X := Test(A,B,X);
```

Ein weiteres Beispiel, das aus einer rekursiven Funktion zur Berechnung von Fibonacci-Zahlen besteht:

```
function Fib(N: Integer): Extended;
begin
  if N = 0 then Fib := 0.0
   else if N = 1 then Fib := 1.0
   else Fib := Fib(N-1) + Fib(N-2);
end;
```

Jeder Aufruf dieser Funktion mit einem Argument größer 8 bringt den Stack des 8087 zum Überlaufen. Der Grund: die Berechnung der letzten Zuweisung erfordert die Speicherung eines Zwischenergebnisses (*Fib(N-1)*) – jede Rekursion belegt also einen Speicherplatz auf dem Stack, der beim neunten Aufruf überläuft. Durch eine korrekte Programmierung wird dieses Problem vermieden:

```
function Fib(N: Integer): Extended;
var
  F1, F2: Extended;
begin
  if N = 0 then Fib := 0.0
   else if N = 1 then Fib := 1.0
   else begin
          F1 := Fib(N-1); F2 := Fib(N-2);
          Fib := F1 + F2;
        end;
end;
```

Hier werden die Zwischenergebnisse als lokale Variablen auf dem Stack des 8086 gespeichert (der natürlich auch irgendwann einmal überläuft – dazu sind aber recht große Verschachtelungstiefen nötig).

Die Ausgabe von Realzahlen

Weil Turbo Pascal bei der Verwendung eines 8087 grundsätzlich im Format *Extended* arbeitet, geben *Write* und *Writeln* nicht zwei, sondern vier Ziffern für den Exponenten aus, um dem erweiterten Rechenbereich gerecht zu werden. Aus demselben Grund erzeugt die Standard-Prozedur *Str* Strings mit vier Stellen für den Exponenten.

Units und der 8087

Units, die den 8087 verwenden, können nur dann in Programme aufgenommen werden, wenn auch die Compilierung des Hauptmoduls im Modus `$N+` stattfindet. Turbo Pascal prüft allerdings bei der Aufnahme einer Unit unabhängig von darin enthaltenen `$N`-Befehlen, ob tatsächlich 8087-Code enthalten ist – falls nein, spielt der Stand des Schalters *O/C/Numeric processing* keine weitere Rolle.

Ermittlung des 8087

Windows bzw. die Emulatorbibliothek WIN87EM.DLL überprüfen automatisch, ob ein Coprozessor installiert ist, und benutzen ihn gegebenenfalls. Ist er nicht vorhanden, wird er durch Routinen in WIN87EM.DLL emuliert. Über die Funktion *GetWinFlags* (definiert in der Unit *WinProcs*) kann mittels der Bitmaske *wf_80x87* (in der Unit *WinTypes* definiert) festgestellt werden, ob ein Coprozessor installiert ist:

```
if GetWinFlags and wf_80x87 <> 0 then
  Writeln('80X87 installiert') else
  Writeln('80x87 nicht installiert');
```

Maschinenprogramme und der Emulator

Die Simulation des Coprozessors durch den Emulator ist nicht auf Pascal-Routinen beschränkt – sie funktioniert auch für (via {$L <Dateiname>} eingebundene) Maschinenprogramme mit direkten 8087-Befehlen. Die einzige Voraussetzung ist ein zusätzlicher Kommandozeilen-Parameter für den Assembler bei der Übersetzung: bei einem Aufruf von TASM wie

```
TASM /E TEST;
```

prüft der Assembler, ob der Quelltext 8087-Befehle enthält – wenn ja, werden diese Befehle speziell gekennzeichnet und damit für den Linker von Turbo Pascal verfügbar gemacht.

III. Turbo Pascal Internas

16. Interne Details

Dieses Kapitel beschreibt detailliert die Speicherbelegung durch Turbo-Pascal-Programme. Weiterhin werden die Attribute der Code-Segmente, das automatische Datensegment, das interne Format der Datentypen, der Heapmanager und der direkte Zugriff auf den Speicher beschrieben.

Code-Segmente

Jedes Modul (das Hauptprogramm oder die Bibliothek und jede Unit) in einem Turbo-Pascal-Programm oder einer DLL verfügt über ein eigenes Code-Segment. Ein solches Code-Segment kann nicht größer als 64 KByte sein, die Gesamtgröße des Programmcodes ist dagegen nur durch den verfügbaren Arbeitsspeicher beschränkt.

Segmentattribute

Jedem Code-Segment können bestimmte Attribute zugeordnet werden. Diese Attribute legen das Verhalten des Code-Segments fest, wenn es in den Speicher geladen wird.

Moveable oder Fixed
Hat ein Code-Segment das Attribut *Moveable*, kann Windows das Segment im physikalischen Speicher hin- und herbewegen, um so den Speicherreservierungen anderer Programme zu entsprechen. Hat das Code-Segment dagegen das Attribut *Fixed*, wird es *nicht* im physikalischen Speicher bewegt. Verwenden Sie das Attribut *Moveable*, wenn es nicht absolut notwendig ist, daß sich ein Code-Segment immer an derselben Adresse im Speicher befindet (zum Beispiel wenn es einen Interrupt-Handler enthält).

Preload oder Demandload
Ein Code-Segment, das mit dem Attribut *Preload* versehen ist, wird automatisch geladen, wenn ein Programm oder eine Bibliothek aktiviert wird. Das

Attribut *Demandload* verzögert das Laden des Segments solange, bis eine Routine im Segment tatsächlich aufgerufen wird.

Discardable oder Permanent

Wenn ein Segment das Attribut *Discardable* trägt, kann Windows den vom Segment besetzten Speicherplatz freigeben, wenn es weiteren Speicherplatz reservieren muß. Ist das Segment dagegen mit dem Attribut *Permanent* versehen, bleibt es immer im Speicher. Wenn ein Programm einen Aufruf in ein *Discardable*-Segment macht, das sich nicht mehr im Speicher befindet, wird dieses Segment zuerst aus der .EXE-Datei geladen. Das dauert länger als wenn das Segment permanent im Speicher geblieben wäre, das Programm benötigt aber zur Ausführung weniger Speicherplatz.

Allgemein ausgedrückt ähnelt ein *Discardable*-Segment in einem Windows-Programm einem Overlay-Segment eines DOS-Programms.

Verändern von Attributen

Die voreingestellten Attribute eines Code-Segments sind *Moveable*, *Preload* und *Permanent*. Diese Attrubute können über den Compiler-Befehl **$C** verändert werden, zum Beispiel:

```
{$C MOVEABLE DEMANDLOAD DISCARDABLE}
```

Weitere Hinweise zum Compiler-Befehl **$C** finden Sie in Kapitel 21 »Compiler-Befehle«.

In einem Windows-Programm besteht keine Notwendigkeit für einen separaten Overlay-Manager. Der Windows-Speichermanager enthält einen Reihe von Funktionen zur Verwaltung von Overlays, die durch die Code-Segment-Attribute gesteuert werden. Diese Funktionen sind jedem Windows-Programm zugänglich.

Das automatische Datensegment

Jedes Programm und jede Bibliothek verfügt über ein sogenanntes automatisches Datensegment, das bis zu 64 KByte groß sein kann. Auf das automatische Datensegment wird immer über das DS-Register (Datensegment-Register) zugegriffen. Es ist in vier Teile aufgeteilt.

Der ersten 16 Bit des automatischen Datensegments enthalten immer den *Task-Header*, in dem Windows verschiedene Systeminformationen speichert.

Der *statische Daten-Bereich* enthält alle globalen Variablen und typisierten Konstanten, die vom Programm oder der Bibliothek deklariert wurden.

Der *Stack* wird zur Speicherung von lokalen Variablen verwendet, die von Prozeduren und Funktionen reserviert wurden. Beim Einsprung in ein Programm werden das Stacksegment-Register (SS) und Stackpointer (SP) so gesetzt,

daß SS:SP auf das erste Byte nach dem Stackbereich im automatischen Datensegment zeigt. Wenn Prozeduren und Funktionen aufgerufen werden, wird SP nach unten bewegt, um Platz für Parameter, die Rücksprungadresse und lokale Variablen zu reservieren. Endet die Routine, wird der Vorgang umgekehrt, indem SP bis auf den Wert vor dem Aufruf erhöht wird. Die voreingestellte Größe des Stackbereichs im automatischen Datensegment ist 8 KByte, kann aber über den Compiler-Befehl **$M** geändert werden.

Im Unterschied zu einem Programm hat eine Bibliothek keinen Stackbereich in ihrem automatischen Datensegment. Wenn ein Aufruf an eine Prozedur oder Funktion in einer DLL erfolgt, zeigt das Register DS in das automatische Datensegment der Bibliothek, das Registerpaar SS:SP wird jedoch nicht verändert. Deswegen verwendet eine Bibliothek immer den Stack des aufrufenden Programms.

Der letzte Teil im automatischen Datensegment ist der lokale Heap. Er speichert alle dynamischen Daten, für die mit der Windows-Funktion *LocalAlloc* Speicher reserviert wurde. Die voreingestellte Größe des lokalen Heap-Abschnitts ist 8 KByte. Dies kann jedoch über den Compiler-Befehl **$M** geändert werden.

In Windows ist es möglich, daß das automatische Datensegment beweglich (*moveable*) ist. Turbo Pascal unterstützt dies aber nicht. Das automatische Datensegment einer Turbo-Pascal-Anwendung oder Bibliothek wird immer gesperrt. Dadurch wird sichergestellt, daß sich der Selektor (die Segment-Adresse) niemals ändert. Wenn das Programm im Standard- oder erweiterten 386er-Modus läuft, hat das keine nachteiligen Auswirkungen, da ein Segment denselben Selektor behält, selbst wenn es im physikalischen Speicher bewegt wird. Wenn allerdings im Real-Modus Windows den lokalen Heap erweitern soll, wird dies eventuell nicht möglich sein, da das automatische Datensegment nicht bewegt werden kann. Wenn Ihr Programm den lokalen Heap verwendet und im Real-Modus laufen soll, müssen Sie sicherstellen, daß die anfängliche Größe des lokalen Heaps (mit dem Befehl **$M** eingestellt) ausreicht, um alle notwendigen Speicherreservierungen zu befriedigen.

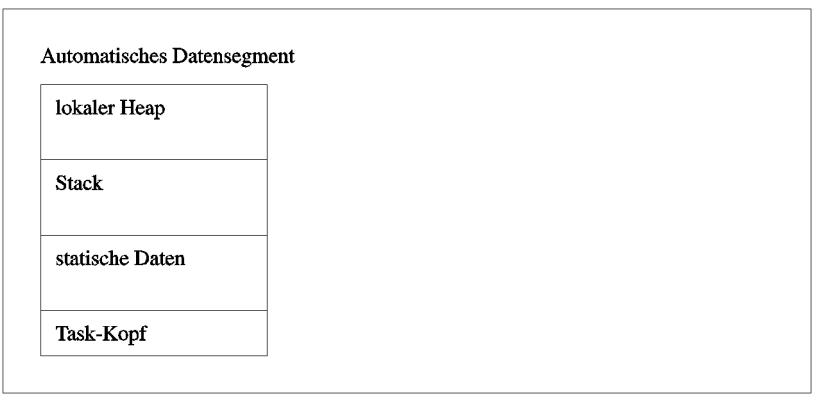


Abbildung 16.1 Das automatische Datensegment

Der Heap-Manager

Windows unterstützt dynamische Speicherbelegung auf zwei verschiedenen Heapbereichen: dem *globalen Heap* und dem *lokalen Heap*. Weitere Informationen zum globalen und lokalen Heap finden Sie in Windows-Programmierhandbuch.

Der globale Heap ist ein Speicherbereich, der allen Programmen zur Verfügung steht. Obwohl globale Speicherblöcke beliebiger Größe belegt werden können, ist der globale Heap für »große« Speicherblöcke gedacht (256 Bytes und mehr). Jeder globale Speicherblock belegt zumindest 20 Byte. Im Standard- und erweiterten 386er-Modus von Windows können maximal 8192 globale Speicherblöcke vergeben, von denen einem Programm nur einige wenige zur Verfügung stehen.

Der lokale Heap ist ein Speicherbereich, der nur Ihrem Programm oder Ihrer Bibliothek zur Verfügung steht. Er befindet sich im oberen Teil des Datensegments des Programms oder der Bibliothek. Die Gesamtgröße der lokalen Speicherblöcke, die auf dem lokalen Heap angelegt werden können, ist 64 KByte minus der Größe des Stacks der Anwendung und ihrer statischen Daten. Aus diesem Grund ist der lokale Heap am besten für »kleine« Speicherblöcke geeignet (256 Byte und weniger). Die voreingestellte Größe des lokalen Heaps ist 8 KByte, dies kann aber mit dem Compiler-Befehl **$M** geändert werden.

Turbo Pascal für Windows enthält einen Heap-Manager, der die Standardfunktionen *New, Dispose, GetMem* und *FreeMem* implementiert. Die Verwaltung des Heaps über *Mark* und *Release*, die in der DOS-Version zur Verfügung stehen, ist nicht möglich. Der Heap-Manager verwendet den globalen Heap für alle Speicherbelegungen. Auf diesem globalen Heap können aber nur maximal 8192 Speicherblöcke reserviert werden, und das ist sicherlich weniger als manche Programm benötigen. Der Heap-Manager in Turbo Pascal umgeht diese Beschränkung, indem er größere Speicherblöcke reserviert und diese bei Bedarf dann in kleinere unterteilt.

Dabei geht der Heap-Manager auf folgende Weise vor: Muß ein sehr großer Speicherbereich reserviert werden, beschränkt er sich darauf, über die Windows-Funktion *GlobalAlloc* diesen Block anzufordern. Müssen dagegen kleine Speicherbereiche reserviert werden, reserviert der Heap-Manager zunächst einen großen Block auf dem globalen Heap und vergibt disen Block in den geforderten kleinen »Portionen«. Ist der Block schließlich vergeben, reserviert der Heap-Manager einen weiteren Bereich und verteilt auch diesen entsprechend den Speicheranforderungen.

Die Variable *HeapLimit* legt die Grenze zwischen einem »kleinen« und einem »großen« Speicherblock fest. Voreingestellt ist sie auf 1024 Bytes. Unterschreitet eine Speicheranforderung diesen Wert, dann reserviert der Heap-Manager einen Speicherblock, dessen Größe durch den Wert der Variablen *HeapBlock* bestimmt wird, zur Weiterverteilung. Voreingestellt ist eine Größe von 8192 Bytes. Im Normalfall gibt es keinen Grund, die Werte für *HeapLimit* und *HeapBlock* zu verändern. Sollte dies dennoch notwendig werden, so müssen Sie darauf achten,

daß der Wert von *HeapBlock* mindestens viermal so groß ist, wie der von *HeapLimit*.

Globale Speicherblöcke, die der Heap-Manager reserviert, tragen immer das Attribut *Fixed*, wodurch sichergestellt ist, daß sich die Selektoren (Segment-Adressen) der Blöcke nicht ändern. Im Standard- und erweiterten 386er-Modus von Windows können auch Blöcke mit dem Attribut *Fixed* immer noch im physikalischen Speicher bewegt werden, um Platz für die jeweiligen Speicheranforderungen zu schaffen. Daher ergeben sich in diesen Modi keinerlei Leistungseinbußen durch den Heap-Manager von Turbo Pascal. Im Real-Modus darf jedoch ein Speicherblock mit dem Attribut *Fixed* nicht im physikalischen Speicher verschoben werden. Wenn Ihr Programm im Real-Modus laufen muß, sollten Sie die Speicherverwaltungsfunktionen von Windows verwenden, wenn Sie dynamische Speicherblöcke reservieren.

Die Variable HeapError

zeigt normalerweise auf die in Turbo Pascal eingebaute Fehlerbehandlung – wenn *New* oder *GetMem* ihre Aufgabe nicht ordnungsgemäß erfüllen können, führen sie einen Aufruf über *HeapError* aus. Die Funktion, auf die *HeapError* standardmäßig zeigt, ist folgendermaßen deklariert:

```
function HeapFunc(Size: Word): Integer; far;
```

Hinweis: Die **far**-Direktive legt fest, daß *HeapFunc* als *far* aufgerufen wird.

Die Installation dieser Routine geschieht durch eine explizite Zuweisung an *HeapError*:

```
HeapError := @HeapFunc;
```

HeapFunc bekommt im Fehlerfall die Größe des zu belegenden Bereichs als Parameter *Size* übergeben und kann nur drei verschiedene Ergebnisse haben:

- der Wert 0 erzeugt einen Laufzeitfehler und damit den sofortigen Abbruch des Programms;
- der Wert 1 steht zwar ebenfalls für eine Fehlerbedingung, bricht das Programm aber nicht ab, sondern bewirkt, daß *New* und *GetMem* den Zeigerwert **nil** zurückliefern;
- der Wert 2 steht für »Wiederholung«. *New* und *GetMem* versuchen in diesem Fall erneut, ihre Aufgabe auszuführen (was einen weiteren Aufruf über *HeapError* zur Folge haben kann).

Die in Turbo Pascal eingebaute Variante von *HeapFunc* liefert immer das Ergebnis 0, bewirkt also den Abbruch des Programms durch einen Laufzeitfehler. Sie läßt sich beispielsweise durch die folgende einfache Routine ersetzen, die für die meisten Anwendungen bereits ausreichend ist:

```
function HeapFunc(Size: Word): Integer; far
begin
  HeapFunc := 1;
end;
```

Nach der Installation dieser Version über eine Zuweisung an *HeapError* liefern *New* und *GetMem* im Fehlerfall den Zeigerwert **nil** zurück, anstatt das Programm abzubrechen.

Interne Datenformate

Integer-Typen

Das von Turbo Pascal verwendete Format zur Darstellung eines Integerwertes hängt von den unteren und oberen Grenzen des erforderlichen Wertebereichs ab:

- Wenn beide Grenzen innerhalb des Bereichs -128..127 liegen (Datentyp *ShortInt*), dann wird ein einzelnes Byte zur Speicherung verwendet, dessen höchstwertiges Bit als Vorzeichen dient.
- Wenn beide Grenzen innerhalb des Bereichs 0..255 liegen (Datentyp *Byte*), dann verwendet Turbo Pascal ein einzelnes Byte zur Speicherung, dessen Wert grundsätzlich als positiv angenommen wird (vorzeichenlos).
- Wenn beide Grenzen innerhalb des Bereichs -32768..32767 liegen (Datentyp *Integer*), dann werden zwei Bytes zur Speicherung verwendet, das höchstwertige Bit des Bytes mit der höheren Adresse dient als Vorzeichen.
- Wenn beide Grenzen innerhalb des Bereichs 0..65535 liegen (Datentyp *Word*), dann verwendet Turbo Pascal zwei Bytes, deren Wert grundsätzlich als positiv angenommen wird.
- In allen anderen Fällen wird ein *LongInt* (vier Bytes, höchstwertiges Bit = Vorzeichen) zur Speicherung verwendet.

Char-Typen

Werte des Typs *Char* oder Teilbereiche davon werden als vorzeichenloses Byte (also wie der Datentyp *Byte*) gespeichert.

Die Typen Boolean, WordBool und LongBool

Werte des Typs *Boolean* werden mit einem einzelnen Byte, des Typs *WordBool* mit einem Word und des Typs *LongBool* mit einem LongInt gespeichert.

Die Typen *Boolean*, *WordBool* und *LongBool* können einen Wert von 0 (= *False*) oder einen Wert ungleich Null (= *True*) annehmen.

Hinweis: In der DOS-Version von Turbo Pascal gibt es nur den Typ *Boolean*, der nur die Werte 0 (= *False*) oder 1 (= *True*) annehmen kann. Aus diesem Grund kann ein mit der Windows-Version von Turbo Pascal erstelltes Programm ohne weiteres Boolesche Felder einer Datendatei lesen, die von einem Programm angelegt wurde, das mit der DOS-Version von Turbo Pascal compiliert wurde. Der umgekehrte Weg ist jedoch nicht möglich.

Aufzählungs-typen

werden als vorzeichenloses Byte (wie der Datentyp *Byte*) gespeichert, wenn die Aufzählung weniger als 256 verschiedene Werte umfaßt. Ansonsten verwendet Turbo Pascal das Speicherformat *Word*.

Real-Typen

speichern die binäre Repräsentation einer Fließkommazahl in drei voneinander getrennten Teilen: dem Vorzeichen (+ oder -), dem *Exponenten* und der *Mantisse*. Diese Unterteilung gilt für sämtliche Real-Typen, also *Real*, *Single*, *Double*, *Extended* und *Comp*. Allgemein läßt sich dieses Format so darstellen:

$$+/- \text{ Mantisse} * 2^{\text{Exponent}}$$

Der Dezimalpunkt wird dabei nicht eigens gespeichert, sondern implizit angenommen: Er steht grundsätzlich rechts des höchstwertigen Bits (MSB = »most significant bit«) - die Mantisse hat also immer einen Wert im Bereich von größer/gleich 0 und kleiner als 2.

Hinweis: In allen Real-Formaten wird das höchstwertige Byte (das den impliziten Dezimalpunkt enthält) auf der höchsten Adresse gespeichert. Darauf folgen die restlichen Bytes der Mantisse in Richtung absteigender Adressen, schließlich kommt der Exponent (in dem zusätzlich das Vorzeichen gespeichert ist).

Der Datentyp Real
belegt 6 Bytes (48 Bit), die folgendermaßen angeordnet sind:

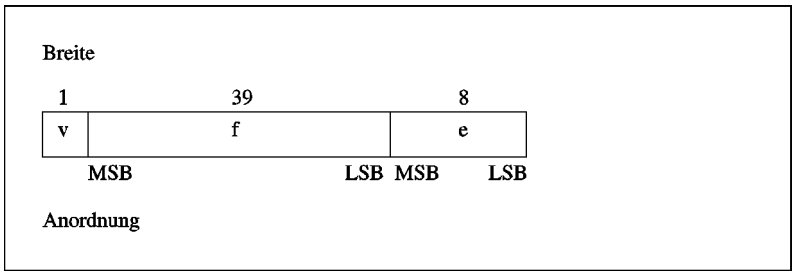


Abbildung 16.2 Das Format des Datentyps Real

Der Wert *w* einer in diesem Format gespeicherten Zahl läßt sich folgendermaßen berechnen:

```
if 0 < e <= 255   then  w = (-1)^v * 2(^e-129) * (1.f)
if e = 0          then  w = 0
```

Hinweis: Im Format *Real* können weder NaNs (Not a Number« = undefinierte Werte) noch denormalisierte Zahlen oder Unendlichkeiten gespeichert werden: denormalisierte Zahlen werden gleich 0 gesetzt, NaNs und Unendlichkeiten erzeugen einen Überlauf.

Der Datentyp Single
belegt 4 Bytes (32 Bit), die folgendermaßen angeordnet sind:

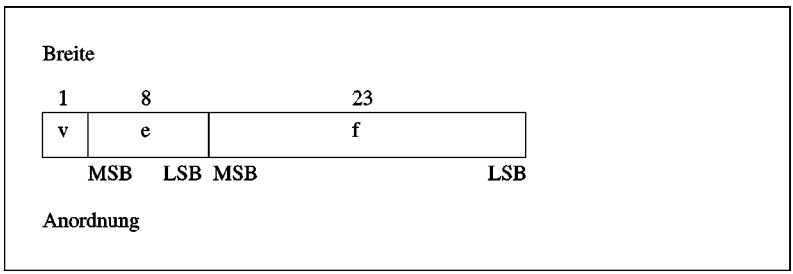


Abbildung 16.3 Das Format des Datentyps Single

Der Wert *w* einer in diesem Format gespeicherten Zahl läßt sich folgendermaßen berechnen:

```
if 0 < e and e < 255   then  w = (-1)^v * 2(^e-127) * (1.f)
if e = 0   and f <> 0  then  w = (-1)^v * 2(^-126) * (0.f)
if e = 0   and f = 0   then  w = (-1)^v * 0
if e = 255 and f = 0   then  w = (-1)^v * Inf
if e = 255 and f <> 0  then  w = NaN
```

Der Datentyp Double
belegt 8 Bytes (64 Bit), die folgendermaßen angeordnet sind:

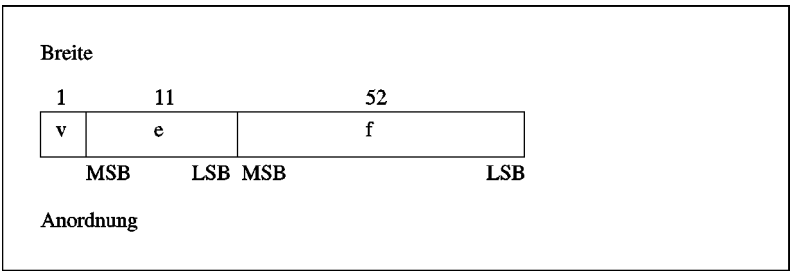


Abbildung 16.4 Das Format des Datentyps Double

Der Wert *w* einer in diesem Format gespeicherten Zahl läßt sich folgendermaßen berechnen:

```
if 0 < e and e < 2047   then  w = (-1)^v * 2(^e-1023) * (1.f)
if e = 0    and f <> 0  then  w = (-1)^v * 2(^-1022) * (0.f)
if e = 0    and f = 0   then  w = (-1)^v * 0
if e = 2047 and f = 0   then  w = (-1)^v * Inf
if e = 2047 and f <> 0  then  w = NaN
```

Der Datentyp Extended
belegt 10 Bytes (80 Bit), die folgendermaßen angeordnet sind:

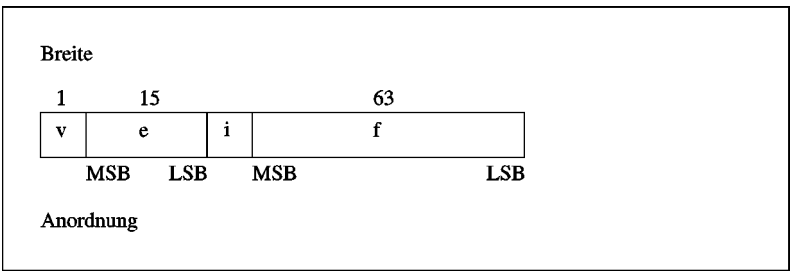


Abbildung 16.5 Das Format des Datentyps Extended

Der Wert *w* einer in diesem Format gespeicherten Zahl läßt sich folgendermaßen berechnen:

```
if 0 <= e and e < 32767 then  w = (-1)^v * 2(^e-16383) * (i.f)
if e = 32767 and f = 0  then  w = (-1)^v * Inf
if e = 32767 and f  0 then  w = NaN
```

Der Datentyp Comp
belegt 8 Bytes (64 Bit), die folgendermaßen angeordnet sind:

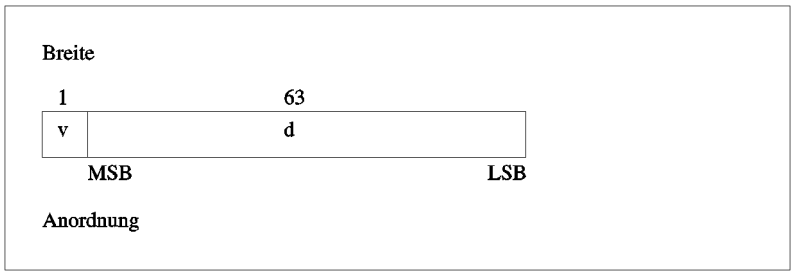


Abbildung 16.6 Das Format des Datentyps Comp

Der Wert *w* einer in diesem Format gespeicherten Zahl läßt sich folgendermaßen berechnen:

```
if v = 1 and d = 0       then  w = NaN
```

Ansonsten ist *w* der durch die 64 Bit im Zweierkomplement angegebene Wert.

Zeiger-Typen

werden mit zwei aufeinanderfolgenden Wörtern (Datentyp *Word*) gespeichert: auf der niedrigeren Adresse steht der Offset-Anteil, auf der höheren Adresse der Segment-Anteil. Die Zeiger-Konstante **nil** hat den Wert $0000:$0000, also gilt Longint(**nil**) = 0.

String-Typen

Ein String wird als Folge einzelner Bytes (Datentyp *Char*) gespeichert, jedes Zeichen belegt also ein Byte. Turbo Pascal stellt die momentane Länge eines Strings in einem weiteren Byte voran, der Gesamtbedarf zur Speicherung beträgt deshalb »maximale Stringlänge plus 1« Bytes. Das Längenbyte steht auf der niedrigsten Speicheradresse, darauf folgen die momentan in dem String gespeicherten Zeichen. Sowohl die Länge als auch die gespeicherten Werte werden als vorzeichenlos betrachtet. Aus der Tatsache, daß die momentane Länge eines Strings in einem Byte gespeichert wird, ergibt sich auch seine Maximallänge: sie beträgt 255 Zeichen.

Mengen

werden als Bit-Array gespeichert, wobei die Position des jeweiligen Bits für die Ordinalität und sein Wert für das Vorhandensein des entsprechenden Elements steht (1« = Element vorhanden, »0« = nicht vorhanden). Eine Menge kann maximal 256 Elemente enthalten und benötigt maximal 32 Bytes Speicherplatz (32 * 8 = 256). Der tatsächliche Speicherplatzverbrauch läßt sich mit der folgenden Formel berechnen:

```
ByteZahl := (Max div 8) - (Min div 8) + 1;
```

Max steht hier für das Element mit der höchsten, *Min* für das mit der niedrigsten Ordinalzahl. Die Position eines Elements mit der Ordinalzahl *E* innerhalb der Menge ergibt sich aus:

```
ByteNummer := (E div 8) - (Min div 8);
BitNummer := E mod 8;
```

Array-Typen

werden als kontinuierliche Folge ihrer Elemente gespeichert, wobei das Element mit dem niedrigsten Index auf der niedrigsten Adresse steht. Mehrdimensionale Arrays werden so gespeichert, daß die Elemente der »äußersten rechten« (also letzten) Dimension direkt aufeinander folgen.

Record-Typen

werden als kontinuierliche Folge ihrer Felder gespeichert, wobei das bei der Definition zuerst aufgeführte Feld auf der niedrigsten Adresse steht. Variante Teile belegen jeweils denselben Speicherbereich, wobei auch hier in der Reihenfolge der einzelnen Komponenten gearbeitet wird.

Datei-Typen

werden intern als Records dargestellt. Typisierte und untypisierte Dateien belegen jeweils 128 Bytes Speicherplatz, die folgendermaßen strukturiert sind:

```
type
  { typisierte und untypisierte Dateien }
  FileRec = record
              Handle:   Word;
              Mode:     Word;
              RecSize:  Word;
```

```
            Private:  array[1..26] of Byte;
            UserData: array[1..16] of Byte;
            Name:     array[0..79] of Char;
          end;
```

Textdateien belegen jeweils 256 Bytes und haben das folgende Format:

```
type
  { Textdateien }
  TextBuf = array[0..127] of Char;
  TextRec = record
            Handle:   Word;
            Mode:     Word;
            BufSize:  Word;
            Private:  Word;
            BufPos:   Word;
            BufEnd:   Word;
            BufPtr:   ^TextBuf;
            OpenFunc: Pointer;
            InOutFunc: Pointer;
            FlushFunc: Pointer;
            CloseFunc: Pointer;
            UserData: array[1..16] of Byte;
            Name:     array[0..79] of Char;
            Buffer:   TextBuf;
          end;
```

Wenn die Datei offen ist, enthält das Feld *Handle* die von DOS vergebene Datei-Referenznummer. Das Feld *Mode* enthält einen der folgenden »magischen« Werte (die *nicht* von 0..3 durchnumeriert sind):

```
const
  fmClosed   = $D7B0;   { Datei ist geschlossen }
  fmInput    = $D7B1;   { nur Lesen }
  fmOutput   = $D7B2;   { nur Schreiben }
  fmInOut    = $D7B3;   { Lesen und Schreiben erlaubt }
```

Der Wert *fmInOut* ist für Textdateien nicht möglich - nur typisierte und untypisierte Dateien können gleichzeitig gelesen und beschrieben werden. Alle anderen Werte bedeuten, daß die entsprechende Datei nicht offen ist - *Mode* ist in diesen Fällen nicht weiter definiert.

Das Feld *UserData* wird von Turbo Pascal und DOS nicht benutzt und steht dem Programmierer für eigene Zwecke zur Verfügung.

Das Feld *Name* enthält den Namen der Datei - allerdings nicht als String mit einem vorangestellten Längenbyte, sondern im Format ASCIIZ: einer beliebigen Folge von Zeichen, die mit einem NUL (ASCII-Code 0) abgeschlossen ist.

Typisierte und untypisierte Dateien enthalten im Feld *RecSize* die Recordlänge (d.h. die Größe einer Komponente); das Feld *Private* wird zwar nicht direkt verwendet, ist aber »reserviert«.

Textdateien enthalten im Feld *BufPtr* einen Zeiger auf einen Puffer mit der durch *BufSize* in Bytes angegebenen Größe. *BufPos* enthält den Index des nächsten zu schreibenden bzw. zu lesenden Zeichens innerhalb dieses Puffers, *BufEnd* die Gesamtzahl der momentan im Puffer gespeicherten Zeichen. Die Zeiger *OpenFunc*, *InOutFunc*, *FlushFunc* und *CloseFunc* zeigen auf die Routine, über die die jeweilige Funktion (*Open*, *Close* usw.) ausgeführt wird. Der Abschnitt »Text-Gerätetreiber« in diesem Kapitel setzt sich näher damit auseinander.

Prozedurale Typen

speichern die Adresse der ihr zugeordneten Routinen mit zwei aufeinanderfolgenden Wörtern (Datentyp *Word*): auf der niedrigeren Adresse steht der Offset-Anteil, auf der höheren Adresse der Segment-Anteil.

Direkte Zugriffe auf Speicher-Adressen

Turbo Pascal stellt drei vordefinierte Arrays zur Verfügung, die direkte Zugriffe auf einzelne Speicheradressen und Ports (I/O-Adressen) ermöglichen:

Die Arrays Mem, MemW und MemL

Mem ist als *array of Byte*, *MemW* als *array of Word* und *MemL* als *array of LongInt* definiert. Alle drei haben eine spezielle Notation ihrer Indizes gemeinsam: zwei Ausdrücke des Typs *Word*, die durch einen Doppelpunkt voneinander getrennt sind. Einige Beispiele:

```
Mem[$0040: $0049] := 7;
Wert := MemW[Seg(V) : Ofs(V)];
LangWert := MemL[64: 3*4];
```

Der erste Befehl setzt die Speicherzelle mit der Adresse $0040:$0047 auf den Wert 7, der zweite liest die ersten beiden Bytes der Variablen *V* und ordnet sie der Integer-Variablen *Wert* zu – falls es sich bei *V* ebenfalls um einen Integer-Typ handelt, ist er mit dem Befehl *Wert := V* identisch. Der dritte Befehl schließlich liest die Speicherzellen $0040:$000C..$0040:$000F und ordnet ihren Wert der Variablen *LangWert* zu, die den Typ *LongInt* haben muß.

Obwohl Turbo Pascal den direkten Zugriff auf den Speicher erlaubt, ist das unter Windows nicht unbedingt ratsam. Sie sollten es Windows überlassen, sich um die Verwaltung des Speichers zu kümmern, wenn Sie nicht riskieren wollen, daß Ihr Programm abstürzt.

17. Objekte

Dieses Kapitel beschreibt die interne Darstellung von Objekten, die Aufrufkonventionen von Methoden und die Einbindung von Methoden, die in Assembler geschrieben wurden.

Das interne Datenformat von Objekten

Das interne Datenformat eines Objekts ist dem eines Records sehr ähnlich. Die Felder eines Objekts werden in der Reihenfolge der Deklaration als eine zusammenhängende Folge von Variablen gespeichert. Jedes Feld, das von einem Vorfahren vererbt wird, wird vor den Feldern gespeichert, die vom Nachkommen neu definiert werden.

Wenn ein Objekttyp virtuelle Methoden, Konstruktoren oder Destruktoren definiert, dann reserviert der Compiler ein zusätzliches Feld im Objekttyp. Dieses 16-Bit-Feld, das als *Feld der virtuellen Methodentabelle* (VMT-Feld) bezeichnet wird, dient dazu, den Offset der VMT der entsprechenden Methode im Datensegment zu speichern. Das VMT-Feld steht direkt hinter den normalen Feldern des Objekttyps. Wenn ein Objekttyp virtuelle Methoden, Konstruktoren oder Destruktoren erbt, erbt er auch das VMT-Feld, so daß kein zusätzliches VMT-Feld reserviert wird.

Initialisiert wird das VMT-Feld einer Instanz von den Konstruktoren des Objekttyps. Ein Programm initialisiert das VMT-Feld niemals selbst und greift auch nicht auf das VMT-Feld zu.

Die folgenden Beispiele veranschaulichen die internen Datenformate von Objekttypen.

```
type
  LocationPtr = ^Location;
  Location = object
    X, Y: Integer;
    procedure Init (PX, PY: Integer);
    function GetX: Integer;
    function GetY: Integer;
  end;
```

```
PointPtr = ^Point;
Point = object(Location)
  Color: Integer;
  constructor Init (PX, PY, PColor: Integer);
  destructor Done; virtual;
  procedure Show; virtual;
  procedure Hide; virtual;
  procedure MoveTo(PX, PY: Integer); virtual;
end;

CirclePtr = ^Circle;
Circle = object(Circle)
  Radius: Integer;
  constructor Init(PX, PY, PColor, PRadius: Integer);
  procedure Show; virtual;
  procedure Hide; virtual;
  procedure Fill; virtual;
end;
```

Abbildung 17.1 enthält den Aufbau der Instanzen von *Location, Point* und *Circle*. Jedes Kästchen stellt eine Speicherstelle dar.

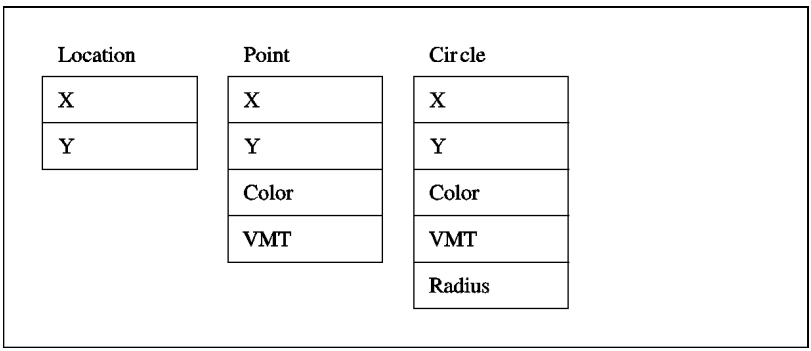


Abbildung 17.1 Aufbau der Instanzen von Location, Point und Circle.

Da *Point* in der Hierarchie der erste Datentyp mit virtuellen Methoden ist, folgt das VMT-Feld direkt hinter dem Feld *Color*.

Virtuelle Methodentabellen

Jeder Objekttyp, der virtuelle Methoden, Konstruktoren oder Destruktoren enthält oder erbt, ist mit einer Tabelle virtueller Methoden (VMT) verbunden, die im initialisierten Abschnitt des Datensegments gespeichert wird. Es gibt nur eine VMT pro Objekttyp (und nicht für jede einzelne Instanz), aber zwei verschiedene Objekttypen können eine VMT niemals gemeinsam verwenden, gleichgültig wie ähnlich sie sich auch sein mögen. VMTs werden vom Compiler

automatisch erstellt. Sie dürfen auch niemals direkt von einem Programm aus verändert werden. Genauso werden die Zeiger auf die VMTs automatisch in den Instanzen des Objekttyps vom Konstruktor des Objekttyps gespeichert und dürfen von einem Programm niemals direkt manipuliert werden.

Das erste Word einer VMT enthält die Größe der Instanzen des damit verbundenen Objekttyps. Diese Information wird von anderen Konstruktoren und Destruktoren verwendet, um festzustellen, wieviele Bytes über die erweiterte Syntax der Standardprozeduren *New* und *Dispose* reserviert oder freigegeben werden sollen.

Das zweite Word einer VMT enthält die negativen Größen der Instanzen des damit verbundenen Objekttyps. Diese Information wird mit Hilfe eines Überprüfungsmechanismus während des Aufrufs von Methoden verwendet, um nicht initialisierte Objekte (Instanzen, die ohne einen Aufruf eines Konstruktors, vorgenommen wurden) und die Konsistenz der VMT zu prüfen. Wenn die Überprüfung der Aufrufe virtueller Funktionen (mit {$R+}) eingeschaltet ist, dann erzeugt der Compiler einen Aufruf der VMT-Überprüfungsfunktion vor jedem Aufruf einer virtuellen Funktion. Die VMT-Überprüfungsfunktion schaut nach, ob das erste Word einer VMT ungleich Null ist, und ob die Summe des ersten und des zweiten Words Null ergibt. Wenn eine dieser Bedingungen nicht zutrifft, wird der Laufzeitfehler 210 erzeugt.

Achtung*:* Das Einschalten der Bereichsüberprüfung und die Überprüfung des Aufrufs virtueller Funktionen verlangsamt Ihr Programm. Das Programm wird etwas größer, so daß Sie {$R+} nur beim Debuggen einschalten und beim fertigen Programm auf {$R-} umschalten sollten.

Das dritte Word einer VMT speichert den Datensegment-Offset der *dynamischen Methodentabelle* (DMT) des Objekttyps oder enthält 0, wenn dieser Objekttyp keine dynamischen Methoden definiert.

Das vierte Word einer VMT ist reserviert und immer auf 0 gesetzt.

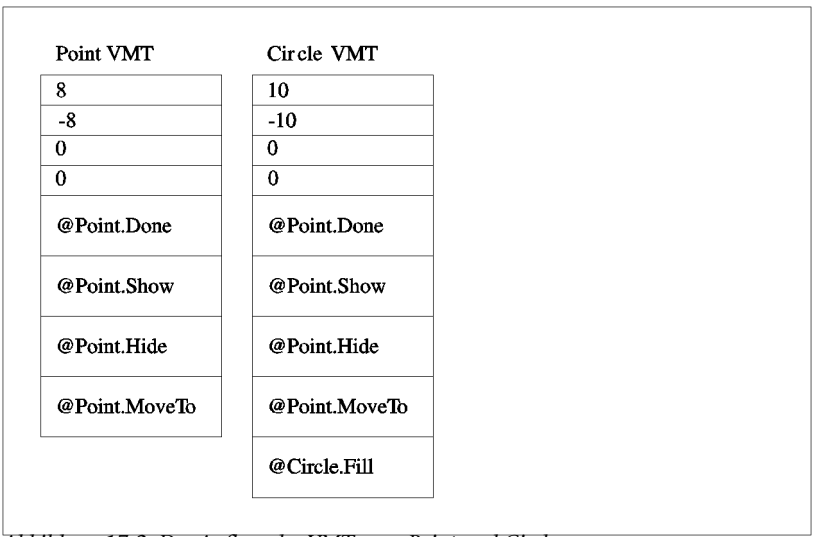


Abbildung 17.2 Der Aufbau der VMTs von Point und Circle

Schließlich kommt beim Offset 8 in der VMT eine Liste von 32-Bit-Zeigern zu den entsprechenden Methoden, ein Zeiger für jede Methode im Objekttyp in der Reihenfolge der Deklaration. Jeder Eintrag enthält die Adresse des entsprechenden Einsprungspunkts der virtuellen Methode.

Abbildung 17.2 zeigt den Aufbau der VMTs der Datentypen *Point* und *Circle* (der Datentyp *Location* hat keine VMT, da er keine virtuellen Methoden, Konstruktoren oder Destruktoren enthält). Jedes kleine Kästchen entspricht einem Word im Speicher und jedes größere Kästchen entspricht zwei Words.

Achten Sie darauf, wie *Circle* die Methoden *Done* und *MoveTo* von *Point* erbt und die Methoden *Show* und *Hide* ersetzt.

Wie bereits früher erwähnt wurde, enthalten die Konstruktoren eines Objekttyps einen speziellen Programmcode, der den Offset der VMT des Objekts in der zu initialisierenden Instanz ablegt. Wenn zum Beispiel eine Instanz *P* eines Punkts und die Instanz *C* eines Kreises vorliegt, dann wird bei einem Aufruf von *P.Init* automatisch der Offset der VMT von *Point* im VMT-Feld von *P* gespeichert und bei einem Aufruf von *C.Init* in gleicher Weise der Offset der VMT von *Circle* im VMT-Feld von *C* gespeichert. Diese automatische Initialisierung ist ein Teil des Codes, der vom Konstruktor angesprungen wird, so daß das VMT-Feld *Self* bereits richtig gesetzt ist, wenn das Programm zum ersten **begin** des Anweisungsteils im Konstruktor kommt. Ein Konstruktor kann also im Bedarfsfall die virtuellen Methoden aufrufen.

Dynamische Methodentabellen

Die VMT eines Objekttyps enthält für jede virtuelle Methode, die dieser Objekttyp definiert, einen vier Bytes langen Eintrag, einen Zeiger auf die virtuelle Methode. Wenn nun Vorfahren des Objekts eine große Zahl virtueller Methoden besitzen, kann die Initialisierung eines abgeleiteten Objekts ziemlich viel Speicher belegen. Auch wenn dieser abgeleitete Typ nur wenige der geerbten virtuellen Methoden überschreibt, enthält die VMT dieses abgeleiteten Objekttyps Zeiger auf alle geerbten virtuellen Methoden, ob sie nun überschrieben wurden oder nicht.

Dynamische Methoden vermeiden diese Verschwendung von Speicherplatz. Turbo Pascal für Windows definiert ein neues Format für Methodentabellen und beschreitet einen neuen Weg der dynamischen Bindung von Methodenaufrufen. Anstatt einen Zeiger auf *jede* Methode, die dynamisch gebunden werden muß, zu speichern, nimmt eine *dynamische Methodentabelle (DMT)* nur jene Methoden auf, die durch ein Objekttyp *überschrieben* wurden. Wenn abgeleitete Objekte nur einige wenige einer großen Zahl geerbter virtueller Methoden überschreiben, dann belegt eine dynamische Methodentabelle weit weniger Speicher als eine VMT.

Die folgenden Objekttypen zeigen die Definition dynamischer Methoden:

```
type
  TBase = object
    X: Integer;
```

```
  constructor Init;
  destructor done; virtual;
  procedure P10; virtual 10;
  procedure P20; virtual 20;
  procedure P30; virtual 30;
  procedure P40; virtual 40;
end;

TDerived = object(TBase)
  Y: Integer;
  constructor Init;
  destructor done; virtual;
  procedure P10; virtual 10;
  procedure P30; virtual 30;
  procedure P50; virtual 50;
end;
```

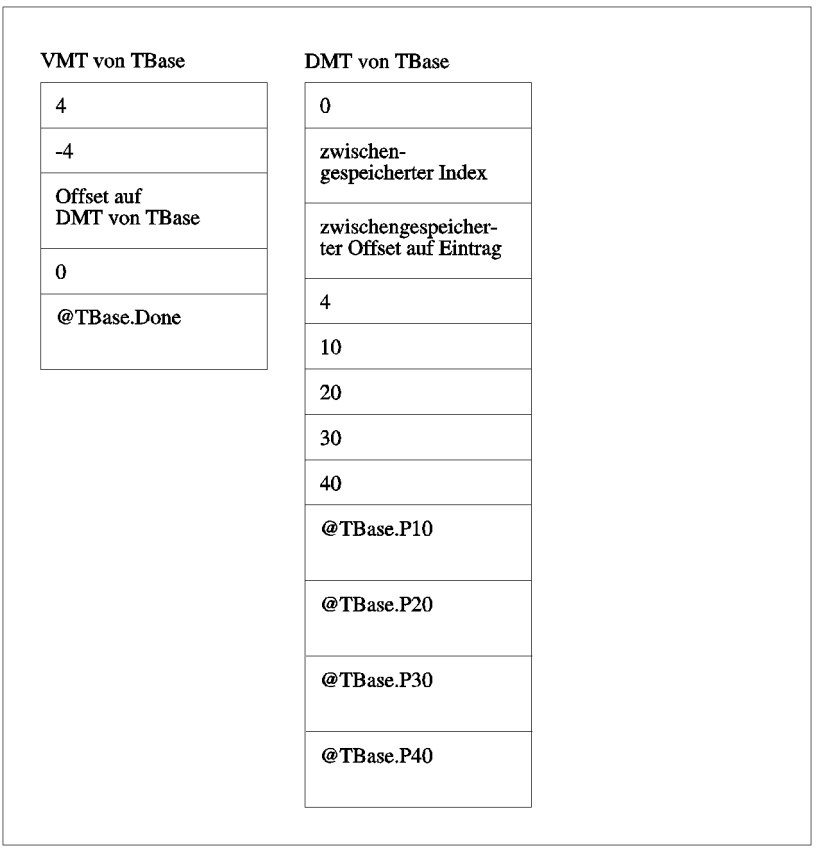


Abbildung 17.3 Der Aufbau der VMT und der DMT von TBase

Die Abbildungen 17.3 und 17.4 zeigen den Aufbau der VMT und DMT sowohl von *TBase* als auch von *TDerived*. Jedes kleine Kästchen stellt eine Word-Speicherstelle, jedes größere stellt zwei Word-Speicherstellen dar.

Ein Objekt hat nur dann eine dynamische Methodentabelle, wenn es dynamische Methoden definiert oder geerbte dynamische Methoden überschreibt. Wenn es dynamische Methode erbt, diese aber nicht überschreibt und selbst auch keine neuen dynamischen Methoden definiert, dann übernimmt es einfach die DMT seines Vorfahren.

Auch die dynamischen Methodentabellen werden wie die VMTs im initialisierten Abschnitt des Datensegments gespeichert.

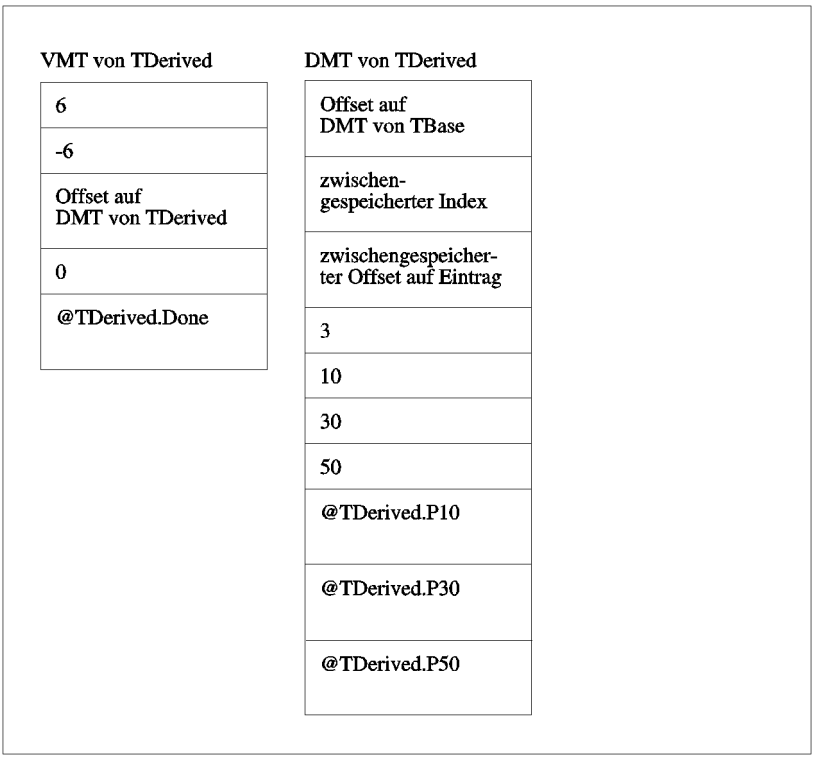


Abbildung 17.4 Der Aufbau der VMT und der DMT von TDerived

Das erste Word einer DMT speichert den Datensegment-Offset der DMT des Vorfahren oder enthält 0, wenn ein solcher nicht existiert.

Das zweite und das dritte Word werden dazu benutzt, Verweise auf dynamische Methoden zwischenzuspeichern. Im Abschnitt »Der Aufruf von dynamischen Methoden« gehen wir genauer auf die Verwendung dieser Felder ein.

Das vierte Word speichert die Zahl der Einträge in der DMT. Diesem Zähler folgt eine entsprechend lange Liste von Indexeinträgen, von denen jeder ein Word belegt. Dieser Liste schließt sich wieder eine ebenso lange Liste von Zeigern auf die korrespondierenden Methoden an.

Die Standardfunktion SizeOf

Wenn die Standardfunktion *SizeOf* auf eine Instanz eines Objekttyp mit einer VMT angewendet wird, dann liefert *SizeOf* die Größe, die in der VMT gespeichert ist. Für Objekttypen, die eine VMT haben, erhält man also mit *SizeOf* die *tatsächliche* Größe der Instanz und nicht die *deklarierte* Größe.

Die Standardfunktion TypeOf

Turbo Pascal definiert eine Standardfunktion, *TypeOf*, die einen Zeiger auf die VMT eines Objekttyps liefert. *TypeOf* erhält einen einzigen Parameter. Das kann entweder die Bezeichnung eines Objekttyps oder eine Instanz eines Objekttyps sein. In beiden Fällen ist das Ergebnis vom Typ *Pointer* ein Zeiger auf die VMT des Objekttyps. *TypeOf* darf nur auf Objekttypen mit einer VMT angewendet werden. Bei allen anderen Datentypen liefert die Funktion einen Fehler.

Die Funktion *TypeOf* kann man zum Überprüfen des tatsächlichen Datentyps einer Instanz verwenden. Zum Beispiel:

```
if TypeOf(Self) = TypeOf(Point) then ...
```

Der Aufruf von virtuellen Methoden

Um eine virtuelle Methode aufzurufen, erzeugt der Compiler Code, um die VMT-Adresse aus dem VMT-Feld im Objekt zu lesen, und führt dann den Aufruf der Methode über den zugehörigen Eintrag aus. Mit einer Variable *PP* des Typs *PointPtr* erzeugt der Aufruf *PP^.Show* zum Beispiel den folgenden Code:

```
les   di,PP             ; PP in ES:DI laden
push  es                ; als Self-Parameter übergeben
push  di
mov   di,es:[di+6]      ; VMT-Offset aus dem VMT-Feld holen
call  DWORD PTR [di+8]  ; VMT-Einsprungspunkt für Show aufrufen
```

Die Kompatibilitätsregeln für Objekttypen erlauben, daß *PP* auf *Point* oder *Circle* oder einen anderen Nachkommen von *Point* zeigt. Und wenn Sie die hier dargestellten VMTs untersuchen, dann sehen Sie, daß bei *Point* der Einsprungspunkt beim Offset 8 in der VMT auf *Point.Show* zeigt, während er bei *Circle* auf *Circle.Show* zeigt. Je nachdem, welcher Datentyp von *PP* zur Laufzeit tatsächlich vorliegt, ruft der *CALL*-Befehl entweder *Point.Show* oder *Circle.Show*, oder die Methode *Show* eines anderen Nachkommens von *Point* auf.

Wenn *Show* eine statische Methode gewesen wäre, dann wäre für *PP.Show* folgender Code erzeugt worden:

```
les   di,PP          ; PP in ES:DI laden
push  es             ; als Self-Parameter übergeben
push  di
call  Point.Show     ; direkter Aufruf von Point.Show
```

Hier spielt es keine Rolle, wohin *PP* zeigt, im Code wird immer die Methode *Point.Show* aufgerufen.

Der Aufruf von dynamischen Methoden

Die Bindung einer dynamischen Methode ist um einiges komplizierter und zeitaufwendiger als die einer virtuellen Methode. Eine virtuelle Methode wird über einen CALL-Befehl über den Zeiger auf die Methode aufgerufen. Dieser Zeiger ist an einem feststehenden Offset in der VMT zu finden. Dagegen müssen beim Aufruf einer dynamischen Methode die DMTs sowohl des Objekttyps als auch die seiner Vorfahren nach dem »aktuellsten« Index der Methode durchsucht werden. Ist diese Suche erfolgreich, wird sie über den korrespondierenden Zeiger aufgerufen. Diese Operationen erfordern weit mehr Befehle als ein Aufruf über eine VMT. Daher übernimmt eine spezielle Routine der Laufzeitbibliothek diese Arbeit.

Nehmen wir an, daß die Methode *Show* des Objekts *Point* als dynamische Methode mit dem Index 200 deklariert worden sei. Mit einer Variable *PP* des Typs *PointPtr* erzeugt der Aufruf *PP^.Show* dann den folgenden Code:

```
les   di,PP             ; PP in ES:DI laden
push  es                ; als Self-Parameter übergeben
push  di
mov   di,es:[di+6]      ; VMT-Offset aus dem VMT-Feld holen
mov   ax,200            ; Methoden-Index in AX laden
call  Dispatch          ; Einsprungspunkt mittels Routine der
                        ; Laufzeitbibliothek ermitteln
```

Die Zuordnungsroutine der Laufzeitbibliothek liest zunächst aus der VMT den Offset auf die DMT und überprüft dann anhand des ersten Zwischenspeicherungsfeldes (dieses Feld speichert den Index der Methode, die zuletzt aufgerufen wurde) der DMT, ob die aufgerufene Methode die gleiche ist, die zuletzt aufgerufen wurde. Ist das der Fall, wird der Offset auf den entsprechenden Methodenzeiger aus dem zweiten Zwischenspeicherungsfeld gelesen und die Methode dann über den Zeiger aufgerufen. Stellt sich aber heraus, daß der Index der Methode nicht dem zwischengespeicherten Index entspricht, durchsucht die Zuordnungsroutine die DMT, dann auch die DMTs der Vorfahren, bis sie einen Eintrag mit dem gesuchten Index findet. Dieser Index und der Offset auf den entsprechenden Methodenzeiger werden dann in den Zwischenspeicherungsfeldern abgelegt und die Methode aufgerufen. Wenn die Zuordnungsroutine aus irgendeinem Grund keinen passenden Eintrag findet (was eigentlich nur passieren kann, wenn die DMT zerstört wurde), wird das Programm angehalten und der Laufzeitfehler 210 ausgegeben.

Trotz Zwischenspeicherung des Indexes der zuletzt aufgerufenen Methode und trotz des äußerst effizienten Zuordnungsverfahrens benötigt die Bindung einer dynamischen Methode wesentlich mehr Zeit als die Bindung einer virtuellen Methode. Dieser Zeitverlust wird jedoch wettgemacht durch den Gewinn an Speicherplatz, den eine dynamische Methodentabelle erbringt.

Aufrufkonventionen von Methoden

Methoden verwenden dieselben Aufrufkonventionen wie normale Prozeduren und Funktionen, außer daß jede Methode einen zusätzlichen Parameter, *Self*, implizit übergibt, der einem **var**-Parameter mit dem Datentyp der Methode entspricht. Der *Self*-Parameter wird immer als letzter Parameter in Form eines 32-Bit-Zeigers auf die Instanz übergeben, über die die Methode aufgerufen wurde. Wenn zum Beispiel eine Variable *PP* des Typs *PointPtr* definiert ist, dann wird der Aufruf *PP^.MoveTo(10,20)* folgendermaßen codiert:

```
mov   ax,10                   ; 10 in AX laden
push  ax                      ; als PX-Parameter übergeben
mov   ax,20                   ; 20 in AX laden
push  ax                      ; als PY-Parameter übergeben
les   di,PP                   ; PP in ES:DI laden
push  es                      ; als Self-Parameter übergeben
push  di
mov   di,es:[di+6]            ; VMT-Offset aus VMT-Feld laden
call  DWORD PTR [di+16]       ; VMT-Eintrag für MoveTo aufrufen
```

Bei der Rückkehr muß die Methode den *Self*-Parameter vom Stack holen, wie dies bei allen anderen Parametern auch der Fall ist.

Die Methoden verwenden immer FAR-Aufrufe, egal, wie die Compileranweisung `$F` eingestellt ist.

Konstruktoren und Destruktoren

Konstruktoren und Destruktoren verwenden dieselben Aufrufkonventionen wie normale Methoden, mit Ausnahme eines zusätzlichen Parameters, der ein Word umfaßt. Dies ist der *VMT*-Parameter, der direkt vor dem *Self*-Parameter auf den Stack gelegt wird.

Bei den Konstruktoren enthält der VMT-Parameter den VMT-Offset, um das VMT-Feld des *Self*-Parameters bei der Initialisierung des *Self*-Parameters zu speichern.

Außerdem wird ein **nil**-Zeiger im *Self*-Parameter übergeben, wenn ein Konstruktor zum Reservieren eines dynamischen Objekts mit Hilfe der erweiterten Syntax der Standardprozedur *New* aufgerufen wird. Deshalb muß der Konstruktor ein neues dynamisches Objekt anlegen, dessen Adresse bei der Rückkehr des

Konstruktors in DX:AX an das aufrufende Programm zurückgegeben wird. Wenn der Konstruktor das Objekt nicht anlegen kann, wird in DX:AX ein **nil**-Zeiger zurückgegeben (siehe Abschnitt »Fehlerbehandlung in Konstruktoren«).

Wenn schließlich ein Konstruktor mit einem vollständigen Methodenbezeichner aufgerufen wird (d.h. zusammen mit einer Bezeichnung des Objekttyps, der ein Punkt und der Methodenbezeichner folgen), wird der Wert Null als Parameter in die VMT-Tabelle eingetragen. Dies zeigt dem Konstruktor an, daß das VMT-Feld von *Self nicht* initialisiert werden soll.

Für einen Destruktor gibt eine Null als VMT-Parameter einen normalen Aufruf an. Ein von Null verschiedener Wert zeigt an, daß der Destruktor mit der erweiterten Syntax der Standardprozedur *Dispose* aufgerufen wurde. Dies führt beim Destruktor dazu, daß *Self* vor der Rückkehr freigegeben wird (die Größe von *Self* finden Sie im ersten Wort der VMT von *Self*.

Erweiterungen von New und Dispose

Die Standardprozeduren *New* und *Dispose* (siehe Kapitel 24 »Verzeichnis der Routinen«) wurden erweitert, damit man einen Aufruf eines Konstruktors bzw. Destruktors als zweiten Parameter zum Anlegen oder Freigeben einer dynamischen Objektvariable optional angeben kann. Die Syntax lautet:

```
New(P, Konstrukt)
```

und

```
Dispose(P, Destrukt)
```

wobei *P* eine Zeigervariable ist, die auf einen Objekttyp zeigt, und *Konstrukt* und *Destrukt* Aufrufe von Konstruktoren und Destruktoren eines Objekttyps sind. Bei *New* bedeutet die neue Syntax nichts anderes als:

```
New(P);
P^.Konstrukt;
```

und für *Dispose* ist die erweiterte Syntax das gleiche wie die Ausführung der Anweisungen

```
P^.Destrukt;
Dispose(P);
```

Ohne die Syntaxerweiterung würden solche »Paare« von Aufrufen von *New* gefolgt von einem Konstruktor und einem Destruktor, hinter denen wiederum ein *Dispose* steht, sehr häufig vorkommen. Die erweiterte Syntax verbessert die Lesbarkeit und erzeugt außerdem einen kürzeren und effektiveren Code.

Der nachfolgende Ausschnitt verdeutlicht die Anwendung der erweiterten Syntax von *New* und *Dispose*:

```
var
  SP : StrFieldPtr;
```

```
  ZP : ZipFieldPtr;
begin
  New(SP, Init(1, 1, 25, 'Vorname'));
  New(ZP, Init(1, 2, 5, 'ZIP', 0, 99999));
  SP^.Edit;
  ZP^.Edit;
  ...
  Dispose(ZP, Done);
  Dispose(SP, Done);
end;
```

Eine zusätzliche Erweiterung ermöglicht die Verwendung von *New* als *Funktion*, die eine dynamische Variable eines bestimmten Typs anlegt und einen Zeiger liefert. Die Syntax lautet dann:

```
New(T)
```

oder

```
New(T, Konstrukt)
```

In der ersten Form kann *T* ein beliebiger Zeigertyp sein. In der zweiten Form muß *T* auf einen Objekttyp zeigen. *Konstrukt* muß ein Aufruf eines Konstruktors des entsprechenden Objekttyps sein. In beiden Fällen ist der Typ des Funktionsresultats *T*.

Hier ein Beispiel:

```
var
  F1, F2 : FieldPtr;
begin
  F1 := New(StrFieldPtr, Init(1, 1, 25, 'Vorname'));
  F2 := New(ZIPFieldPtr, Init(1, 2, 5, 'ZIP', 0, 99999));
  ...
  WriteLn(F1^.GetStr);       { Aufruf von StrField.GetStr }
  WriteLn(F2^.GetStr);       { Aufruf von ZIPField.GetStr }
  ...
  Dispose(F2, Done);         { Aufruf von Field.Done }
  Dispose(F1, Done);         { Aufruf von StrField.Done }
end;
```

Beachten Sie, daß *F1* und *F2*, auch wenn sie den Typ *FieldPtr* haben, über die erweiterte Zuweisungskompatibilität von Zeigern einen Zeiger erhalten können, der auf einen Nachkommen von *Field* zeigt. Da *GetStr* und *Done* virtuelle Methoden sind, wird der Auswahlmechanismus für virtuelle Methoden die richtigen Funktionen *StrField.GetStr*, *ZipField.GetStr*, *Field.Done* sowie auch *StrField.Done* aufrufen.

Methoden in Assembler

Methoden, die in Assembler implementiert wurden, können mit dem integrierten Assembler oder mit Hilfe der Compiler-Anweisung $L und des Schlüsselworts **external** in Turbo Pascal eingebunden werden. Die Deklaration einer externen Methode in einem Objekttyp unterscheidet sich nicht von einer normalen Methode. Die Implementation dieser Methode enthält nur den Deklarationskopf der Methode, dem das reservierte Wort **external** folgt.

In einem Assembler-Quelltext wird das Zeichen @ anstelle des Punkts (.) für einen vollständigen Bezeichner angegeben (der Punkt hat nämlich in Assembler bereits eine andere Bedeutung und darf deshalb nicht in einem Variablenbezeichner verwendet werden). Der Pascal-Bezeichner *Rect.Init* wird zum Beispiel in Assembler als *Rect@Init* geschrieben. Die Syntax mit dem @-Zeichen kann man sowohl in *PUBLIC*- als auch in *EXTRN*-Variablenbezeichnern verwenden.

Als Beispiel für eine Methode in Assembler haben wir ein einfaches *Rect*-Objekt implementiert.

```
type
  Rect = object
    X1, Y1, X2, Y2: Integer;
    procedure Init(XA, YA, XB, YB: Integer);
    procedure Union(VAR R: Rechteck);
    function Contains(X, Y: Integer): Boolean;
  end;
```

Ein Objekt *Rect* ist ein Rechteck, das mit Hilfe der vier Koordinaten, *X1*, *Y1*, *X2* und *Y2* dargestellt wird. Die linke obere Ecke des Rechtecks ist durch *X1* und *Y1* festgelegt und die rechte untere Ecke ist durch *X2* und *Y2* definiert. In der Methode *Init* werden den Rechteckkoordinaten Werte zugewiesen. Die Methode *Union* berechnet das kleinste Rechteck, das sowohl das Rechteck selbst als auch ein anderes Rechteck enthält. In der Methode *Contains* wird der Wert *True* zurückgegeben, wenn der angegebene Punkt innerhalb des Rechtecks liegt, und der Wert *False*, wenn das nicht der Fall ist. Andere Methoden, zum Beispiel Verschieben, Verändern der Größe, Berechnen von Schnittflächen und Vergleiche mit anderen Rechtecken, die *Rect* zu einem brauchbaren Objekt komplettieren, lassen sich leicht implementieren.

Die Pascal-Implementationen der Methoden von *Rect* enthalten nur den Deklarationskopf mit dem Schlüsselwort **external**.

```
{$L RECT}

procedure Rect.Init(XA, YA, XB, YB: Integer); external;
procedure Rect.Union(var R: Rect); external;
procedure Rect.Contains(X, Y: Integer): Boolean; external;
```

Es ist natürlich keineswegs erforderlich, daß alle Methoden als externe Unterprogramme deklariert werden. Jede der einzelnen Methoden kann entsprechend den Erfordernissen entweder in Pascal oder in Assembler implementiert werden.

Die Assembler-Quelldatei RECT.ASM, in der die drei externen Methoden implementiert sind, ist nachfolgend abgedruckt:

```
            TITLE     Rect
            LOCALS    @@
  ; Rechteck-Struktur
  Rect      STRUC
  X1        DW        ?
  Y1        DW        ?
  X2        DW        ?
  Y2        DW        ?
  Rect      ENDS
  code      SEGMENT   BYTE PUBLIC
            ASSUME    cs:code
  ; procedure Rect.Init(XA, YA, XB, YB)
            PUBLIC    Rect@Init
  Rect@Init           PROC      FAR
  @XA                 EQU       (WORD PTR [bp+16])
  @YA                 EQU       (WORD PTR [bp+14])
  @XB                 EQU       (WORD PTR [bp+12])
  @YB                 EQU       (WORD PTR [bp+10])
  @Self               EQU       (DWORD PTR [bp+6])
            push      bp        ; BP sichern
            mov       bp,sp     ; Stackaufbau festlegen
            les       di,@Self  ; Self in ES:DI laden
            cld                 ; Zeiger nach oben
            mov       ax,@XA    ; X1 := XA
            stosw
            mov       ax,@YA    ; Y1 := YA
            stosw
            mov       ax,@XB    ; X2 := XB
            stosw
            mov       ax,@YB    ; YB := YB
            stosw
            pop       bp        ; BP wiederherstellen
            ret       12        ; Parameter abräumen, Rückkehr
  Rect@Init           ENDP
  ; procedure Rect.Union(var R: Rect)
            PUBLIC    Rechteck@Union
  Rect@Union          PROC     FAR
  @R                  EQU       (DWORD PTR [bp+10])
  @Self               EQU       (DWORD PTR [bp+6])
            push      bp        ; BP sichern
            mov       bp,sp     ; Stackaufbau festlegen
            push      ds        ; DS sichern
            lds       si,@R     ; R in DS:SI laden
            les       di,@Self  ; Self in ES:DI laden
            cld                 ; Zeiger nach oben
```

```
            lodsw
            scasw
            jge       @@1       ; Falls R.X1 = X1 dann @@1
            dec       di
            dec       di
            stosw               ; X1 := R.X1
@@1:        lodsw
            scasw
            jge       @@2       ; falls R.Y1 >= Y1 dann @@2
            dec       di
            dec       di
            stosw               ; Y1 := R.Y1
@@2:        lodsw
            scasw
            jle       @@3       ; falls R.X2 <= X2 dann @@3
            dec       di
            dec       di
            stosw               ; X2 := R.X2
@@3:        lodsw
            scasw
            jle       @@4       ; falls R.Y2 <= Y2 dann @@4
            dec       di
            dec       di
            stosw               ; Y2 := R.Y2
@@4:        pop       ds        ; DS wiederherstellen
            pop       bp        ; BP wiederherstellen
            ret       8         ; Parameter abräumen, Rückkehr
Rect@Union  ENDP
; function Rect@Contains(X, Y: Integer): Boolean
            PUBLIC    Rect@Contains
Rect@Contains PROC      FAR
@X          EQU       (WORD PTR [bp+12])
@Y          EQU       (WORD PTR [bp+10])
@Self       EQU       (DWORD PTR [bp+6])
            push      bp        ; BP sichern
            mov       bp,sp     ; Stackaufbau festlegen
            les       di,@Self  ; Self in ES:DI laden
            mov       al,0      ; False zurückgeben
            mov       dx,@X
            cmp       dx,es:[di].X1
            jl        @@1       ; if (X < X1) or (X > X2) goto @@1
            cmp       dx,es:[di].X2
            jg        @@1
            mov       dx,@Y
            cmp       dx,es:[di].Y1
            jl        @@1       ; if (Y < Y1) or (Y > Y2) goto @@1
            cmp       dx,es:[di].Y2
            jg        @@1
```

```
          inc       ax
@@1:      pop       bp          ; BP wiederherstellen
          ret       8           ; Parameter abräumen, Rückkehr
Rect@Contains       ENDP
code      ENDS
          END
```

Fehlerbehandlung in Konstruktoren

Wie zu Beginn des Kapitels 16 »Speicherbelegung und Datenformate« im Abschnitt »Die Variable HeapError« beschrieben, können Sie eine Fehlerbehandlungsfunktion für Probleme am Heap mit Hilfe der Variable *HeapError* in der Unit *System* installieren. Diese Möglichkeit beeinflußt auch den Programmablauf in einem Konstruktor.

Standardmäßig wird der Laufzeitfehler 203 erzeugt, wenn bei der Speicherplatzreservierung einer dynamischen Objektinstanz, die mit Hilfe der erweiterten Syntax von *New* vorgenommen wird, nicht mehr genug Speicherplatz vorhanden ist. Wenn Sie eine Heap-Fehlerfunktion installieren, die anstelle des standardmäßigen Funktionsergebnisses von 0 den Wert 1 zurückgibt, dann liefert ein Aufruf eines Konstruktors mittels *New* den Wert **nil**, wenn die Anforderung nicht erfüllt werden kann (statt das Programm abzubrechen).

Der Code zur Speicherplatzreservierung und Initialisierung des VMT-Felds einer dynamischen Instanz ist Teil der Einsprungssequenz eines Konstruktors: Am **begin** des Anweisungsteils des Konstruktors ist die Instanz bereits angelegt und initialisiert. Scheitert die Speicherplatzreservierung und wird von der Heap-Fehlerfunktion eine 1 zurückgegeben, dann übergeht der Konstruktor die Ausführung des Anweisungsteils und gibt einen **nil**-Zeiger zurück. Der in der *New*-Anweisung angegebene Zeiger, mit dem der Konstruktor aufgerufen wurde, wird also auf **nil** gesetzt.

Ist das Programm am **begin** des Anweisungsteils des Konstruktors angelangt, so wurde die Instanz des Objekttyps erfolgreich angelegt und initialisiert. Wenn jedoch der Konstruktor selbst dynamische Variablen anzulegen versucht, um zum Beispiel die Zeigerfelder in einer Instanz zu initialisieren, kann es ebenfalls vorkommen, daß die Reservierung dieses Speicherplatzes scheitert. Wenn das passiert, sollte ein sauber konzipierter Konstruktor die vorgenommenen Reservierungen wieder rückgängig machen und die Instanz des Objekttyps freigeben, so daß ein **nil**-Zeiger zurückgegeben werden kann. Damit dieser »Rückzug« möglich ist, wurde in Turbo Pascal eine neue Standardprozedur, *Fail*, implementiert, die keine Parameter hat und nur von einem Konstruktor aus aufgerufen werden darf. Ein Aufruf von *Fail* führt dazu, daß ein Konstruktor die dynamische Instanz, die vorher beim Einsprung in den Konstruktor angelegt wurde, wieder freigibt, und daß sofort die Rückkehr mit dem Ergebnis **nil** angetreten wird, um den Fehler anzuzeigen.

Wenn dynamische Instanzen mit Hilfe der erweiterten Syntax von *New* angelegt werden, zeigt das Ergebnis **nil** in der entsprechenden Zeigervariable

an, daß die Operation gescheitert ist. Leider gibt es keine solche Zeigervariable, die man nach der Erstellung einer statischen Instanz oder dem Aufruf eines vererbten Konstruktors untersuchen könnte. Stattdessen erlaubt Turbo Pascal den Einsatz eines Konstruktors als Boolean-Funktion in einem Ausdruck: Das Ergebnis *True* steht für eine erfolgreiche Durchführung und *False* zeigt an, daß der Vorgang gescheitert ist, weil die Prozedur *Fail* innerhalb eines Konstruktors aufgerufen werden mußte.

Im folgenden Programm sind zwei einfache Objekttypen mit Zeigern implementiert. Die erste Version des Programms enthält keine Fehlerbehandlung im Konstruktor.

```
type
  LinePtr = ^Line;
  Line = string[79];
  BasePtr = ^Base;
  Base = object
    L1, L2: LinePtr;
    constructor Init(S1, S2: Line);
    destructor Done; virtual;
    procedure Dump; virtual;
  end;
  DerivedPtr = ^Derived;
  Derived = object (Base)
    L3, L4: LinePtr;
    constructor Init(S1, S2, S3, S4: Line);
    destructor Done; virtual;
    procedure Dump; virtual;
  end;

var
  BP: BasePtr;
  DP: DerivedPtr;

constructor Base.Init(S1, S2: Line);
begin
  New(L1);
  New(L2);
  L1^ := S1;
  L2^ := S2;
end;

destructor Base.Done;
begin
  Dispose(L2);
  Dispose(L1);
end;
procedure Base.Dump;
begin
  WriteLn('B: ', L1^, ', ', L2^, '.');
end;
```

```
constructor Derived.Init(S1, S2, S3, S4: Line);
begin
  Base.Init(S1, S2);
  New(L3);
  New(L4);
  L3^ := S3;
  L4^ := S4;
end;

destructor Derived.Done;
begin
  Dispose(L4);
  Dispose(L3);
  Base.Done;
end;

procedure Derived.Dump;
begin
  WriteLn('A: ', L1^, ', ', L2^, ', ', L3^, ', ', L4^, '.');
end;

begin
  New(BP, Init('Turbo', 'Pascal'));
  New(DP, Init('Nord', 'Ost', 'Süd', 'West' );
  BP^.Dump;
  DP^.Dump;
  Dispose(DP, Done);
  Dispose(BP, Done);
end.
```

Das nächste Beispiel zeigt, wie das vorherige Beispiel mit einer Fehlerbehandlungsroutine umgeschrieben werden kann. Die Typen- und Variablendeklarationen bleiben gleich, deshalb werden sie nicht mehr wiederholt.

```
constructor Base.Init(S1, S2: Line);
begin
  New(L1);
  New(L2);
  if (L1 = nil) or (L2 = nil) then
  begin
    Base.Done;
    Fail;
  end;
  L1^ := S1;
  L2^ := S2;
end;

destructor Base.Done;
begin
  if L2  nil then Dispose(L2);
```

```
  if L1  nil then Dispose(L1);
end;

constructor Derived.Init(S1, S2, S3, S4: Line);
begin
  if not Base.Init(S1, S2) then Fail;
  New(L3); New(L4);
  if (L3 = nil) or (L4 = nil) then
  begin
    Derived.Done; Fail;
  end;
  L3^ := S3;
  L4^ := S4;
end;

destructor Derived.Done;
begin
  if L4  nil then Dispose(L4);
  if L3  nil then Dispose(L3);
  Base.Done;
end;

function HeapFunc(Groesse: Word): Integer; far;
begin
  Heapfunc := 1;
end;

begin
  HeapError := @HeapFunc;      { Heap-Fehlerbehandlung installieren }
  New(BP, Init('Turbo', 'Pascal'));
  New(DP, Init('Nord', 'Ost', 'Süd', 'West' );
  if (BP = nil) or (DP = nil) then
    WriteLn('Fehler bei Speicherplatzreservierung');
  else
  begin
    BP^.Dump;
    DP^.Dump;
  end;
  if DP  nil then Dispose(DP, Done);
  if BP  nil then Dispose(BP, Done);
end.
```

Achten Sie darauf, wie die entsprechenden Destruktoren in *Base.Init* und *Derived.Init* eingesetzt werden, um alle vorher vorgenommenen Speicherplatzreservierungen zurückzunehmen, bevor schließlich *Fail* aufgerufen wird, um den Vorgang endgültig abzubrechen. Außerdem sollten Sie beachten, daß der Aufruf von *Base.Init* in *Derived.Init* innerhalb eines Ausdrucks codiert wurde, damit der erfolgreiche Abschluß des vererbten Konstruktors überprüft werden kann.

18. Interne Details

Dieses Kapitel beschreibt die Aufrufkonventionen für Prozeduren und Funktionen, die Übergabe eventueller Parameter und die Bearbeitung von Interrupts. Außerdem wird die Definition eigener Exit-Prozeduren behandelt.

Der Aufruf von Funktionen und Prozeduren

Die Übergabe von Parametern beim Aufruf einer Prozedur oder Funktion geschieht durch den Stack des Prozessors: Die Parameter werden in der Reihenfolge auf den Stack gelegt, die durch ihre Deklaration festgelegt ist, und innerhalb der aufgerufenen Routine wieder vom Stack gelesen bzw. entfernt. Im Prinzip lassen sich die Maschinenbefehle für einen Aufruf wie

```
BeispielProc(X, Y, Z);
```

also folgendermaßen darstellen:

```
PUSH
PUSH Y
PUSH Z
CALL Beispiel Proc
```

Parameter werden entweder als Wert oder als Variablen-Referenz übergeben. Bei einer Wertübergabe wird lediglich der entsprechende Wert auf den Stack gelegt, bei einer Variablen-Übergabe dagegen die Adresse der entsprechenden Variablen.

Variablen-Parameter

werden dann verwendet, wenn die aufgerufene Routine den entsprechenden formalen Parameter als **var** deklariert. Die aufgerufene Routine bekommt einen Zeiger übergeben: sie liest den tatsächlichen Wert selbst, hat also direkten Zugriff auf die entsprechende Variable und kann ihren Wert verändern. Der

übergebene Zeiger entspricht in jedem Fall dem Datentyp *Pointer* (4 Bytes, organisiert als 2 Werte des Typs *Word*).

Wert-Parameter

werden - je nach Größe und Typ - entweder als Wert oder durch einen Zeiger übergeben. Abgesehen von Gleitkommazahlen, bei denen Turbo Pascal (zum einen aus »historischen«, zum anderen aus Kompatibilitätsgründen) ebenfalls eine Wertübergabe verwendet, werden Daten mit 1, 2 oder 4 Bytes Umfang als Werte und alle »größeren« Typen als Zeiger übergeben. Im letzteren Fall kann die Original-Variable trotzdem innerhalb der aufgerufenen Routine nicht verändert werden: Turbo Pascal stellt automatisch einige Befehle voran, die die tatsächlichen Werte in einen lokalen Speicherbereich kopieren, und »vergißt« den übergebenen Zeiger danach.

Hinweis: Da die Prozessoren der 8086-Familie nicht über PUSH- und POP-Befehle für einzelne Bytes verfügen, werden Parameter mit einem Byte ebenfalls als Wort (2 Bytes) übergeben. Der tatsächliche Wert befindet sich in diesem Fall im niederwertigen Byte, das höherwertige Byte ist unbenutzt (und undefiniert).

- Integer-Typen werden im Format ihrer Speicherung übergeben, d.h. je nach Typ mit einem, zwei oder vier Bytes. Für Werte des Typs *LongInt* legt Turbo Pascal das niederwertige Wort zuletzt auf den Stack, das dadurch auch auf der niedrigeren Adresse innerhalb des Stack-Segments steht.

- Char-Typen werden in derselben Weise übergeben wie *Byte*, d.h. als einzelnes und vorzeichenloses Byte.

- Boolesche Werte werden als einzelnes Byte mit dem Wert 0 oder 1 übergeben.

- Aufzählungstypen werden als einzelnes und vorzeichenloses Byte übergeben, wenn die Aufzählung weniger als 256 verschiedene Werte enthält, ansonsten im Format *Word*.

- Parameter im Format *Real* werden – obwohl sie sechs Bytes beanspruchen – als Werte übergeben, um die Kompatibilität zur Version 3.0 zu erhalten.

- Parameter im Format *Single* werden als Werte mit vier Bytes übergeben.

- Dasselbe gilt für alle anderen Fließkommaformate: obwohl *Double*, *Extended* und *Comp* mehr als 4 Bytes Speicherplatz beanspruchen, erfolgt ihre Übergabe als Wert (und damit über den Stack des 8086).

Hinweis: Die Version 4.0 von Turbo Pascal verwendet für *Single..Comp* den Rechenstack des 8087 zur Übergabe und begrenzt damit die Anzahl möglicher Fließkomma-Parameter bei Aufrufen auf 8. Dieser Mechanismus wird aus Gründen der Kompatibilität zu anderen Programmiersprachen nicht mehr benutzt.

- Zeiger-Typen werden als zwei Werte des Typs *Word* übergeben, wobei Turbo Pascal den Offset-Anteil als letztes auf den Stack legt - dieser Wert steht während der Übergabe also auf der niedrigeren Adresse innerhalb des Stack-Segments.

- String-Typen werden grundsätzlich als Zeiger auf den tatsächlichen Speicherplatz übergeben.

- Mengen-Typen werden als Zeiger auf einen Bereich übergeben, der grundsätzlich 32 Bytes umfaßt (innerhalb dieses Bereichs sind auch Teilmengen so gespeichert, als würden sie Platz für 256 Elemente bieten).

- Arrays und Records werden im Format ihrer Speicherung auf dem Stack übergeben, wenn ihre Gesamtgröße zwischen einem und vier Bytes liegt. Ansonsten erfolgen auch Wertübergaben durch Zeiger.

Funktionsergebnisse

Ordinale Ergebnisse (Integer-Typen, *Char*, *Boolean* und Aufzählungstypen) werden über die Register des Prozessors zurückgeliefert: einzelne Bytes in AL, Worte in AX und Doppelworte im Registerpaar DX:AX, wobei DX das höherwertige Wort enthält.

- Ergebnisse im Format *Real* werden durch drei Prozessor-Register zurückgeliefert: das höchstwertige Wort in DX, das mittlere in BX und das niederwertige in AX (DX:BX:AX).

- Ergebnisse in den anderen Real-Formaten (*Single*, *Double*, *Extended* und *Comp*) werden nicht über die Register des Prozessors, sondern über das jeweils oberste Register des 8087 (ST(0), »TOS«) zurückgeliefert.

- Zeiger-Typen werden (wie *LongInt*) über das Registerpaar DX:AX zurückgeliefert: DX enthält das Segment, AX den Offset-Anteil.

- Für String-Typen muß die aufrufende Funktion vor den eigentlichen Parametern einen Zeiger auf einen temporären Speicherbereich auf den Stack legen. Die Funktion liefert ihr Ergebnis in diesem Bereich zurück und darf den Zeiger (im Gegensatz zu allen anderen Parametern) nicht vom Stack entfernen.

Andere Funktionsergebnisse sind nicht erlaubt.

Aufrufe als near und far

Die Prozessoren der 8086-Familie unterscheiden zwischen zwei Arten des Aufrufs bzw. Rücksprungs, die man als *near* und *far* bezeichnet. Aufrufe und Rücksprünge mit *near* laufen innerhalb eines einzigen Code-Segments ab, bei der Codierung als *far* kann mit verschiedenen Segmenten gearbeitet werden.

Ein als *near* codierter Aufruf-Befehl (CALL) legt eine Rücksprungadresse mit 16 Bit auf den Stack, die nur den Offset-Anteil innerhalb des momentanen Code-Segments enthält. Ein als *far* codierter Aufruf (CALL FAR) legt zusätzlich den momentanen Inhalt des Codesegment-Registers (CS) auf dem Stack ab und setzt dieses Register danach neu, benötigt also 32 Bit Speicherplatz. Die entspre-

chenden Rücksprung-Befehle (RET und RET FAR) lesen die auf dem Stack abgelegten Werte und setzen damit die Prozessor-Register (IP bzw. CS:IP).

Abhängig von der Deklaration der Routine wählt Turbo Pascal automatisch die richtige Art des Aufrufs: wenn die Routine im Interface-Teil einer Unit deklariert ist, wird sie als **far** codiert (und aufgerufen). Eine Deklaration innerhalb des Implementations-Teils oder des Hauptprogramms hat dagegen die Codierung und den Aufruf als **near** zur Folge – derartige Prozeduren und Funktionen können nur innerhalb der Unit bzw. des Programms aufgerufen werden, in dem sie selbst deklariert sind.

Es kann natürlich Gründe geben, warum man eine Routine abweichend von diesen generellen Regeln als **near** oder **far** deklariert – wenn der Aufruf beispielsweise über einen Zeiger geschieht, muß die Codierung grundsätzlich als **far** erfolgen (also bei Exit-Prozeduren, Textdatei-Treibern sowie den meisten Fehlerbehandlungsroutinen). Nach dem Befehl {$F+} oder mit Hilfe der **far**-Direktive erzeugt der Compiler immer »weite« Rücksprünge (und merkt sich auch, daß die entsprechenden Routinen als **far** aufgerufen werden müssen). Mit {$F-} (der Standardvorgabe von Turbo Pascal) wird die zuvor beschriebene Automatik wieder aktiviert. Dies ist bei Verwendung der **far**-Direktive nicht notwendig.

Hinweis: Mit {$F-} läßt sich eine »bedingungslose« Codierung als **near** nicht erzwingen – im Interface-Teil einer Unit ist dieser Befehl also wirkungslos.

Eingeschachtelte Routinen

Eine Prozedur oder Funktion wird als »eingeschachtelt« bezeichnet, wenn sie innerhalb einer anderen Prozedur oder Funktion deklariert ist (also lokal dazu). In diesem Fall ist sichergestellt, daß sich aufrufende und aufgerufene Routine im selben Code-Segment befinden – deshalb nimmt der Compiler hier eine weitere Optimierung vor und codiert diese Aufrufe auch dann als near, wenn der Modus {$F+} gesetzt ist.

Beim Aufruf einer lokal deklarierten Prozedur oder Funktion erzeugt der Compiler direkt vor dem CALL den Befehl PUSH BP, übergibt also das BP-Register der aufrufenden Routine als zusätzlichen Parameter. Nachdem die aufgerufene Routine ihren eigenen Stack (mit PUSH BP/MOV BP,SP) eingerichtet hat, kann auf das BP-Register der *aufrufenden* Routine über [BP+4] (**near**) bzw. [BP+6] (**far**) und damit auf die lokalen Variablen dieser übergeordneten Routine zugegriffen werden. (Wenn die aufrufende Routine ihrerseits eingeschachtelt ist, wird die Verbindung zur nächsthöheren Ebene wiederum über [BP+4] bzw. [BP+6] hergestellt usw).

Ein Beispiel dazu:

```
procedure A; near;
var
  IntA: Integer;

  procedure B; far;
  var
    IntB: Integer;
```

```
procedure C; near;
var
  IntC: Integer;
begin
  inline($8B/$46/ /       { MOV AX,[BP+IntC]   ; AX = IntC }
         $8B/$5E/$04/     { MOV BX,[BP+4]      ; BX = B's Stack }
         $36/$8B/$47/ /   { MOV AX,SS:[BX+IntB]; AX = IntB }
         $8B/$5E/$04/     { MOV BX,[BP+4]      ; BX = B's Stack}
         $36/$8B/$5F/$04/ { MOV BX,SS:[BX+6]   ; BX = A's Stack }
         $36/$8B/$47/ A); { MOV AX,SS:[BX+IntA]; AX = IntA }
end   {C};

begin {B}
end;  {B}
begin {A}
end;  {A}
```

Hinweis: Eingeschachtelte Prozeduren und Funktionen können nicht als **external** deklariert werden; die Zuordnung prozeduraler Variablen bzw. ihre Übergabe als prozedurale Parameter ist nicht erlaubt.

Start- und Endcode von Prozeduren und Funktionen

Jede Prozedur oder Funktion von Turbo Pascal beginnt und endet mit einigen standardisierten Befehlen:

```
PUSH BP             ; BP auf den Stack
MOV  BP,SP          ; Stand des Stackzeigers -- BP-Register
SUB  SP, LocalSize  ; Platz für lokale Variablen schaffen
   ....
   ....    { Prozedur- oder Funktionsblock: }
   ....    { Übergebene Parameter und lokale Variablen }
   ....    { werden via BP adressiert }
   ....
   ....    { Für Funktionen: MOV AX, Ergebnis[BP] }
   ....
MOV  SP,BP          ; Stackzeiger zurück - Freigabe lokaler Variablen
POP  BP             ; BP wiederherstellen
RET  ParamSize      ; Parameter vom Stack und Rücksprung
```

LocalSize steht hier für die Menge an Speicherplatz, die für lokale Variablen benötigt wird - wenn die Routine ohne lokale Variablen arbeitet, dann fehlt der SUB-Befehl ganz. Das Register BP hält den Stand des Stackzeigers (SP) zu Beginn der Routine fest und bleibt während der gesamten Ausführung der Prozedur/Funktion bzw. wird vor Veränderungen zwischengespeichert. Wenn die Prozedur als **near** codiert ist, beginnt der als letztes deklarierte Parameter mit der Adresse *BP+04*, bei einer Codierung als **far** mit *BP+06*.

Vor dem Rücksprung wird der ursprüngliche Inhalt des Stackzeigers wieder

hergestellt (und damit der durch lokale Variablen belegte Speicherplatz wieder freigegeben). *ParamSize* steht für die Anzahl der Bytes, die durch übergebene Parameter belegt wurden – sie werden durch den Rücksprungbefehl entfernt, der je nach Art des Aufrufs entweder als RET oder als RET FAR codiert ist.

Die oben angeführten Befehle werden generiert, wenn der Compiler-Befehl **{$W-}** angegeben wurde. Im voreingestellten Modus **{$W+}** erzeugt der Compiler den folgenden Stackrahmen:

```
INC  BP             ; FAR-Merkmal für Windows
PUSH BP             ; BP auf den Stack
MOV  BP,SP          ; Stand des Stackzeigers -- BP-Register
PUSH DS             ; DS auf denh Stack
SUB  SP, LocalSize  ; Platz für lokale Variablen schaffen
   ....
   ....
MOV  SP,BP          ; Stackzeiger zurück - Freigabe lokaler Variablen
                    ; und des DS-Registers
POP  BP             ; BP wiederherstellen...
DEC  BP             ; ...und richtigstellen
RET  ParamSize      ; Parameter vom Stack und Rücksprung
```

Für Prozeduren oder Funktionen, die mit der Direktive **export** compiliert werden, wird folgender Stackrahmen erzeugt:

```
MOV  AX,DS          ; DS-Selektor in AX laden
NOP                 ; Platz für Windows-Flickwerk
INC  BP             ; FAR-Merkmal für Windows
PUSH BP             ; BP auf den Stack
MOV  BP,SP          ; Stand des Stackzeigers -- BP-Register
PUSH DS             ; DS auf den Stack
MOV  DS,AX          ; DS initialisieren
SUB  SP, LocalSize  ; Platz für lokale Variablen schaffen
PUSH SI             ; SI sichern
PUSH DI             ; DI sichern
   ....
   ....
POP  DI             ; DI wiederherstellen
POP  SI             ; SI wiederherstellen
LEA  SP,[BP-2]      ; Freigabe lokaler Variablen
POP  DS             ; DS wiederherstellen
POP  BP             ; BP wiederherstellen...
DEC  BP             ; ...und richtigstellen
RET  ParamSize      ; Parameter vom Stack und Rücksprung
```

Um zwischen *near*- und *far*-Stackrahmen unterscheiden zu können, erwartet Windows für alle als *far* codierten Routinen (eingeschlossen die **export**-Routinen) einen *ungeraden* BP-Wert an [BP+0]. Darüber hinaus erwartet Windows an der Stelle [BP+2] den Selektor für das Datensegment des Moduls, aus dem heraus die Routine aufgerufen wurde. Dies erklärt die Befehlssequenz INC BP, PUSH DS und DEC BP, die Turbo Pascal für **far**- und **export**-Routinen generiert.

Eine **export**-Routine muß den Inhalt der Register SI und DI bewahren. Turbo Pascal generiert die dazu notwendigen PUSH- und POP-Befehle und auch die Sequenz MOV AX,DS und NOP im Eintrittscode solcher Routinen. Diese Sequenz wird von Windows gefordert. Ist die Routine in einem Programm definiert (also eine CallBack-Funktion) wandelt Windows diese Sequenz in drei NOP-Befehle um und bereitet damit diese Routine für den Aufruf durch die Windows-Funktion *MakeProcInstance* vor. Ist die Routine dagegen in einer Bibliothek definiert, dann setzt Windows in diese drei Bytes den Befehl MOV AX, *nnnn* ein, wobei *nnnn* für den Selektor, also die Segment-Adresse des automatischen Datensegments der Bibliothek steht.

Verfügbare Prozessor-Register

Die Register BP, SP, SS und DS müssen während der Ausführung einer Funktion oder Prozedur konstant gehalten bzw. vor Veränderungen zwischengespeichert werden. Bei **export**-Routinen gilt dies auch für die Register SI und DI. Alle anderen Register sind frei verwendbar.

Exit-Prozeduren

Durch die Installation einer Exit-Prozedur können Sie festlegen, wie sich ein Programm nach seinem »offiziellen« Ende verhält. Sinnvoll ist so etwas speziell dann, wenn Sie dafür sorgen müssen, daß bestimmte Aktionen unabhängig von der Art des Programmendes auf jeden Fall ausgeführt werden (wie beispielsweise das Zurücksetzen von Interrupt-Vektoren oder das Schließen von Dateien).

Turbo Pascal deklariert eine globale Zeigervariable namens *ExitProc*, über die bei jeder Art von Programmende ein Aufruf stattfindet - egal, ob das Programm normal, durch *Halt* oder durch einen Laufzeitfehler beendet wird.

Über eine explizite Zuweisung an *ExitProc* können Sie eine eigene Exit-Prozedur festlegen, die zwei Voraussetzungen erfüllen muß: zum einen darf sie keine Parameter erwarten, zum anderen muß sie mit dem Befehl {$F+} compiliert worden sein (also als *far*).

Die Zuweisung der Startadresse dieser Prozedur an *ExitProc* besteht aus zwei Schritten: dem Speichern des vorherigen Wertes und der eigentlichen Zuweisung selbst. Tatsächlich sieht die Laufzeitbibliothek von Turbo Pascal nämlich nicht nur eine Prozedur zum Beenden eines Programms vor, sondern eine *Prozedurkette* - speziell dann, wenn Units mit aufgenommen worden sind, deren Initialisierungs-Teil diverse Systemparameter verändert hat. Diese Units verlassen sich darauf, daß ihre eigene Exit-Prozedur beim Beenden des Programms aufgerufen wird, um Veränderungen (wie Interrupt-Vektoren, direkte Speicherzugriffe etc.) wieder rückgängig zu machen.

Aus diesem Grund werden auch an den Code einer eigenen Exit-Prozedur

noch einige Anforderungen gestellt, die das folgende Programmfragment illustriert:

```
program ExitTest;
var
  ExitSave: Pointer;  { für die originale Exit-Prozedur }

procedure MyExit; far;
begin
  ExitProc := ExitSave;  { ExitProc wieder aufs Original - sollte als
                         * erster Schritt* geschehen }
   ...
   ...
end;

begin  { Hauptprogramm }
  ExitSave := ExitProc;  { Original-Zeiger speichern }
  ExitProc := @MyExit;  { und die eigene Routine installieren }
   ...
   ...
end.
```

Zu Beginn des Programms wird also der ursprüngliche Inhalt von *ExitProc* in einer globalen Variablen gespeichert, darauf folgt die Installation der eigenen Exit-Prozedur. Innerhalb von *MyExit* muß dieser Original-Inhalt auf jeden Fall wieder gesetzt werden, bevor das Programm dann tatsächlich endet.

Die Routine der Laufzeitbibliothek, die für das Beenden von Programmen zuständig ist, führt solange Aufrufe über *ExitProc* aus, bis dieser Zeiger den Wert **nil** bekommt. Damit keine Endlosschleifen entstehen können, wird *ExitProc* vor dem Aufruf jeder Exit-Prozedur auf **nil** gesetzt. Wenn Ihre Prozedur also »vergißt«, den zu Beginn des Programms gespeicherten Wert wiederherzustellen, wird die Kette dadurch abgebrochen.

Ein weiteres Detail: Die Zuweisung an *ExitProc* sollte direkt zu Beginn der Exit-Prozedur stattfinden. Ansonsten können Endlosschleifen entstehen: ein Abbruch der Exit-Prozedur durch einen Laufzeitfehler wird als Programmende gewertet, dadurch die Exit-Prozedur aufgerufen, die wiederum einen Laufzeitfehler erzeugt usw.

Für die gezielte Behandlung von Fehlern gibt es eine weitere Möglichkeit: eine Exit-Prozedur kann den »Grund« des Programmendes über die Variablen *ExitCode* und *ErrorAddr* ermitteln. Bei einem normalen Programmende hat *ExitCode* den Wert 0, bei einem Abbruch durch *Halt* enthält diese Variable das Argument von *Halt* (*ErrorAddr* zeigt in beiden Fällen auf **nil**). Beim Abbruch durch einen Laufzeitfehler enthält *ExitCode* die entsprechende Fehlernummer, *ErrorAddr* zeigt auf den Befehl, der den Fehler verursacht hat.

Die als allerletztes ausgeführte Exit-Prozedur (die durch die Laufzeitbibliothek selbst installiert wird) schließt die Dateien *Input* und *Output*, setzt die durch *System* veränderten Interrupt-Vektoren wieder zurück und prüft *ErrorAddr*: falls hier nicht der Wert **nil** gespeichert ist, wird eine entsprechende Meldung ausgegeben.

Wenn Sie eigene Meldungen für Laufzeitfehler einsetzen wollen, können Sie

eine Exit-Prozedur schreiben, die *ErrorAddr* überprüft, die entsprechende Meldung ausgibt und danach *ErrorAddr* auf **nil** setzt, um weitere Fehlermeldungen zu verhindern.

Und schließlich: Nachdem alle Exit-Prozeduren ausgeführt worden sind, folgt der Rücksprung zu Windows, wobei der Wert von *ExitCode* als Ergebnis des Programms übergeben wird.

Bearbeitung von Interrupts

Der durch den Compiler erzeugte Code und die Routinen der Laufzeitbibliothek sind interruptfähig, d.h. können beliebig durch Interrupts unterbrochen werden. Der größte Teil der Laufzeitbibliothek ist darüber hinaus reentrant - es ist also möglich, Interrupt-Prozeduren in Turbo Pascal zu schreiben.

Interrupt-Prozeduren

werden in Turbo Pascal durch das reservierte Wort **interrupt** deklariert. Der Kopf einer **interrupt**-Prozedur ist standardisiert:

```
procedure  <Name> (Flags,CS,IP,AX,BX,CX,DX,SI,DI,DS,ES,BP: Word);
interrupt;
```

Alle Register des Prozessors werden hier als (Pseudo-)**var**-Parameter übergeben und können innerhalb der Prozedur verändert werden. Es ist möglich, die Parameterliste zu verkürzen - aber nur in einer ganz bestimmten Form:

```
procedure  <Name> (      CS,IP,AX,BX,CX,DX,SI,DI,DS,ES,BP: Word);
interrupt;
procedure  <Name> (            AX,BX,CX,DX,SI,DI,DS,ES,BP: Word);
interrupt;
    .....
    .....
procedure  <Name> (                                  ES,BP: Word);
interrupt;
procedure  <Name> (                                     BP: Word);
interrupt;
```

Anders gesagt: die »vorderen« Parameter dürfen ausgelassen werden, solange die Reihenfolge der restlichen Angaben eingehalten wird. Überzählige Parameter haben meistens einen Systemabsturz zur Laufzeit des Programms zur Folge. Ähnlich katastrophal endet der Versuch, ein oder mehrere »Löcher« in der Liste zu erzeugen, also

```
procedure  <Name> (Flags, CS,       CX,DX,SI,       ES,BP: Word);
interrupt;          { *** FALSCH !!! *** }
```

Vorsicht: Derartige Fehler werden vom Compiler nicht entdeckt.

Unabhängig von der Deklaration der Parameter beginnt der Compiler jede als **interrupt** deklarierte Prozedur mit der Speicherung aller Register auf dem Stack und dem Setzen des DS-Registers auf das Datensegment des Programms:

```
PUSH  AX
PUSH  BX
PUSH  CX
PUSH  DX
PUSH  SI
PUSH  DI
PUSH  DS
PUSH  ES
PUSH  BP
MOV   BP,SP
SUB   SP,LocalSize     ; Platz für lokale Variablen
MOV   AX,SEG DATA      ; Datensegment-Startadresse
MOV   DS,AX            ; Turbo Pascal-Datensegment - DS-Register
```

Einen CLI-Befehl, der weitere Interrupts sperrt, werden Sie hier nicht finden – falls er notwendig ist, müssen Sie ihn (via **inline** oder **asm**) selbst codieren. Der Code, mit dem der Compiler eine **interrupt**-Prozedur beendet, ist ebenfalls standardisiert:

```
MOV   SP,BP            ; Freigabe lokaler Variablen
POP   BP               ; und danach alle Register wieder vom Stack
POP   ES
POP   DS
POP   DI
POP   SI
POP   DX
POP   CX
POP   BX
POP   AX
IRET                   ; Rücksprung
```

Zugriffe auf die Parameter einer **interrupt**-Funktion sprechen die auf dem Stack gespeicherten Werte an: im Prinzip handelt es sich also um eine **var**-Deklaration (auch wenn sie formal nicht als solche ausgewiesen ist). Ein Befehl wie

```
AX := $4421;
```

innerhalb der **interrupt**-Prozedur sorgt also dafür, daß das AX-Register vor dem Rücksprung auf den Wert $4421 gesetzt wird. Auf diese Weise kann auch eine Interrupt-Routine Ergebnisse zurückliefern.

interrupt-Prozeduren, die durch Hardware-Ereignisse (wie z.B. den »Clock Tick« der eingebauten Uhr) ausgelöst werden, müssen ohne die Bibliotheksroutinen für Ein- und Ausgaben und die dynamische Speicherverwaltung auskommen, weil der entsprechende Code nicht reentrant ist. Dasselbe gilt für DOS-Funktionen, denn auch diese sind nicht reentrant.

19. Ein- und Ausgabe

Dieses Kapitel beschreibt, wie Sie eigene Textdatei-Gerätetreiber definieren können und was Sie beachten müssen, wenn Sie auf Ein-/Ausgabe-Adressen zugreifen wollen.

Textdatei-Gerätetreiber

Wie bereits in Kapitel 11 »Die Unit System« erwähnt, erlaubt Turbo Pascal die Definition eigener Gerätetreiber für Textdateien. Ein solcher »Treiber« besteht aus einem Satz von vier Funktionen, die als Schnittstelle zwischen Turbo Pascal und einem beliebigen Gerät dienen. Die vier Funktionen übernehmen die Ausführung der Operationen *Open*, *InOut* (Ein-/Ausgabe), *Flush* (erzwungenes Schreiben eines Pufferinhalts) und *Close*. Alle vier Funktionsköpfe haben dasselbe Format:

```
function  <Name>(var F: TextRec): Integer;
```

Der Datentyp *TextRec* ist in der Unit *WinDos* definiert und wurde bereits im Kapitel »Die Unit WinDos« besprochen. Er kann (allerdings mit äußerster Sorgfalt!) redefiniert werden, wie wir es in den folgenden Beispielen demonstrieren.

Die Namen der vier Funktionen sind beliebig - allen gemeinsam ist, daß sie mit dem Befehl {$F+} oder mit Hilfe der **far**-Direktive compiliert werden, einen **var**-Parameter des Typs *TextRec* (bzw. einer Modifikation davon) erwarten und einen Integer zurückliefern müssen.

Um die Verbindung zwischen dem Treiberprogramm und dem dadurch angesprochenen Gerät herzustellen, müssen Sie eine eigene *Assign*-Prozedur schreiben. Der Name dieser Routine ist ebenfalls beliebig – er sollte eine logische Verbindung zu seiner Funktion darstellen. Eine eigene Version von *Assign* muß die folgenden Schritte ausführen:

- Setzen der vier Zeiger in der Datei-Variablen mit den Adressen der vier Funktionen (*Open*, *InOut*, *Flush* und *Close*).
- Setzen des Feldes *Mode* mit der »magischen« Konstanten *fmClosed*.

- Löschen des Eintrags im Feld *Name*, Speicherung der Größe und der Adresse des (vordefinierten) Puffers in den Feldern *BufSize* bzw. *BufPtr*.

Wenn wir davon ausgehen, daß die vier Funktionen die Namen *DevOpen*, *DevInOut*, *DevFlush* und *DevClose* haben, dann könnte die Prozedur *DevAssign* so aussehen:

```
procedure DevAssign(var F: Text);
begin
  with TextRec(F) do   { Typ-Umwandlung in TextRec! }
    begin
      Mode      := fmClosed;
      BufSize   := SizeOf(Buffer);
      BufPtr    := @Buffer;
      OpenFunc  := @DevOpen;
      InOutFunc := @DevInOut;
      FlushFunc := @DevFlush;
      CloseFunc := @DevClose;
      Name[0]   := #0;
    end;
end;
```

Die vier Funktionen des Gerätetreibers können das Feld *UserData* der Datei-Variablen zur Speicherung eigener Informationen benutzen – dieses Feld wird von Turbo Pascal selbst nicht verwendet.

Die Open-Funktion

wird durch die Standard-Prozeduren *Reset*, *Rewrite* und *Append* aufgerufen. Sie öffnet die mit der Datei-Variablen verbundene Datei bzw. initialisiert ein angeschlossenes Gerät. Vor ihrer Ausführung wird das Feld *Mode* auf einen der drei Werte *fmInput*, *fmOutput* oder *fmInOut* gesetzt und zeigt so an, durch welche Prozedur der Aufruf erfolgt ist.

Open muß grundsätzlich vor allen anderen Funktionen des Gerätetreibers aufgerufen werden. Die Standardprozedur *Assign* nützt diese Tatsache aus: sie initialisiert nur das Feld *OpenFunc* der Datei-Variablen und überläßt es *Open*, die restlichen Funktionszeiger abhängig von *Mode* zu setzen. *InOut*, *Flush* und *Close* müssen daher nicht erst selbst bestimmen, in welcher Weise sie arbeiten sollen.

Die InOut-Funktion

wird durch *Read*, *Readln*, *Write*, *Writeln*, *Page*, *Eof*, *Eoln*, *SeekEof*, *SeekEoln* und *Close* aufgerufen und übernimmt den gesamten Datentransfer zwischen dem Programm und dem angeschlossenen Gerät.

Die Arbeitsweise von *InOut* wird durch *Mode* bestimmt: wenn *Mode* den Wert

fmInput hat, dann liest *InOut* maximal *BufSize* Zeichen in den Puffer der Datei-Variablen (^*BufPtr*) und liefert die Anzahl der gelesenen Zeichen indirekt über *BufEnd* zurück. Nach jedem neuen Setzen des Pufferinhalts mit gelesenen Daten muß sie *BufPos* auf 0 setzen; wenn sie *BufEnd* als Ergebnis einer Eingabeanforderung auf 0 setzt, dann liefern nachfolgende Aufrufe von *Eof* den Wert *True* zurück.

Wenn *Mode* auf den Wert *fmOutput* gesetzt ist, dann schreibt die Funktion *InOut* bei jedem Aufruf die durch *BufPos* angegebene Anzahl von Zeichen zu dem Gerät. Die Daten werden via ^*BufPtr* gelesen; nach dem Beenden der Aktion wird *BufPos* auf 0 zurückgesetzt.

Die Flush-Funktion

wird automatisch nach jedem *Read*, *Readln*, *Write*, *Writeln* oder direkt (durch die Standard-Prozedur *Flush*) aufgerufen. Ihre Arbeitsweise ist ebenfalls von *Mode* abhängig:

Wenn *Mode* den Wert *fmInput* hat, dann setzt *Flush* die Felder *BufPos* und *BufEnd* auf 0, ignoriert also sämtliche Zeichen, die sich eventuell noch im Puffer befinden. (Diese Möglichkeit ist allerdings selten praktisch verwendbar.)

Wenn *Mode* den Wert *fmOutput* hat, dann schreibt *Flush* (auf dieselbe Weise wie *InOut*) den Inhalt des Puffers und stellt so sicher, daß sämtliche vom Programm ausgegebenen Zeichen ohne Verzögerung zum angeschlossenen Gerät gesendet werden.

Die durch *Flush* durchgeführten Operationen sind optional; falls diese Funktion lediglich als »Dummy« definiert ist, werden Ausgaben erst dann physikalisch geschrieben, wenn der Puffer voll ist oder die Datei geschlossen wird.

Die Close-Funktion

wird durch die Standard-Prozedur *Close* aufgerufen und schließt d
die Datei. (*Reset*, *Rewrite* und *Append* rufen *Close* ebenfalls auf, wenn die Datei zum entsprechenden Zeitpunkt bereits offen ist.)

Wenn *Mode* den Wert *fmOutput* hat, ruft das Dateisystem von Turbo Pascal vor *Close* zuerst einmal *InOut* auf, um eventuell noch im Puffer befindliche Zeichen zu schreiben.

Direkte Zugriffe auf I/O-Adressen

Die Arrays Port und PortW

Port und *PortW* sind als **array of Byte** bzw. **array of Word** deklariert und erlauben direkte Zugriffe auf die Ein-/Ausgabeadressen des Systems. Beide Arrays sind eindimensional, über den Index (Typ *Word*) wird die entsprechende I/O-Adresse festgelegt. Zuordnungen von Werten zu Komponenten der Arrays *Port* und *PortW* haben eine Ausgabe (OUT-Befehl) zur entsprechenden I/O-Adresse zur Folge, Bezüge auf Komponenten dieser Arrays innerhalb eines Ausdrucks führen eine Leseaktion (IN-Befehl) durch. Einige Beispiele:

```
Port[$20] := $FF;                  { schreibt zur I/O-Adresse $20 }
Port[Base] := Port[Base] xor Maske;
while Port[$B2] and $80 do ....  { warten }
Index := $15
PortW[Index] := 0;     { schreibt zur I/O-Adresse Index }
```

Port und *PortW* können nur gelesen und beschrieben werden - die Verwendung einzelner Komponenten als **var**-Parameter oder eine Bezugnahme auf das gesamte Array (d.h. ohne Index) ist nicht erlaubt.

20. Automatische Optimierung

Turbo Pascal optimiert den erzeugten Code in mehrfacher Hinsicht automatisch, wobei sich die angewendeten Verfahren in verschiedene Klassen einteilen lassen. Die folgenden Abschnitte beschreiben die Einzelheiten.

Vorausberechnung von Konstanten

Konstanten-Ausdrücke, die sich aus ordinalen Werten zusammensetzen, werden von Turbo Pascal durch den Compiler berechnet: eine Zuweisung wie

```
X := 3 + 4 * 2;
```

erzeugt deshalb exakt denselben Code wie

```
X := 11;
```

Dasselbe gilt nicht nur für (ordinale) Argumente der Funktionen *Abs*, *Sqr*, *Succ*, *Pred*, *Odd*, *Lo*, *Hi* und *Swap*, bei denen der Compiler gleich das Ergebnis einsetzt, sondern auch für das Verbinden von Strings:

```
XStr:= 'Ein ' + 'Text';
```

erzeugt denselben Code wie

```
XStr := 'Ein Text';
```

Indizierungen von Arrays, die über Konstanten definiert sind, werden vom Compiler als absolute Adressen berechnet: ein Ausdruck wie *Index[5,5]* erzeugt also denselben Code wie der Zugriff auf eine einfache Variable.

Gemeinsame Speicherung mehrfach verwendeter Konstanten

String-Konstanten, die mehrfach innerhalb eines Programms angegeben sind, werden nur einmal gespeichert. Ein Beispiel:

```
if (ch = 'E') then begin
   Writeln('Programm beendet.');
   Writeln('Auf Wiedersehen!');
end
else if (ch = 'X') then begin
   Writeln('Programm abgebrochen...');
   Writeln('Auf Wiedersehen!');
end;
```

Hier reserviert der Compiler für die Zeichenfolge *Auf Wiedersehen!* nur einmal Speicherplatz – der Aufruf von *Writeln* in der vorletzten Zeile enthält lediglich einen weiteren Verweis auf die Adresse, die bereits in der dritten Programmzeile verwendet wird. Der Compiler vergleicht allerdings nur vollständige Strings miteinander – die Zeichenfolge *Programm* wird im gezeigten Beispiel also zweimal gespeichert.

Auswertung zusammengesetzter boolescher Ausdrücke

Turbo Pascal unterstützt zwei Arten der Code-Erzeugung für die Auswertung zusammengesetzter Ausdrücke mit den Operatoren **and** und **or**: vollständige Berechnung und das Kurzschlußverfahren:

- Bei einer vollständigen Berechnung werden sämtliche Teile des Ausdrucks berechnet - selbst dann, wenn das Ergebnis bereits feststeht, also beispielsweise der erste Teil einer **and**-Operation den Wert *False* ergibt. Diese Art der Auswertung ist z. B. erforderlich, wenn Teile des Ausdrucks aus Funktionsaufrufen bestehen, über die weitere Operationen ausgeführt und/oder Werte verändert werden (Seiteneffekt).

- Das Kurzschlußverfahren garantiert eine strikte Auswertung von links nach rechts und bricht ab, sobald das Ergebnis des Gesamtausdrucks feststeht - beispielsweise dann, wenn der erste Teil einer **or**-Operation den Wert *True* ergibt. Diese Art der Auswertung erhöht nicht nur die Arbeitsgeschwindigkeit des Programms und erzeugt normalerweise kompakteren Code, sondern ermöglicht auch Konstrukte, die bei vollständiger Berechnung nicht zulässig wären. Zwei Beispiele dazu:

```
while (I <= Length(S)) and (S[I] <> ' ') do Inc(I);
while (P <> nil) and (P^.Wert <> 5) do P := P^.Next;
```

Hier wird in beiden Fällen beim Kurzschlußverfahren die zweite Prüfung nicht durchgeführt, wenn die erste den Wert *False* ergibt. Bei einer vollständigen Berechnung ergäben sich Zugriffe auf nicht definierte Adressen als Teil der Prüfung selbst (vgl. Kapitel 21 »Compiler-Befehle«).

Welche Art von Code der Compiler zur Auswertung zusammengesetzter Bedingungen generiert, wird über den Schalter `$B` kontrolliert, der standardmäßig auf `$B-` (Kurzschlußverfahren) gesetzt ist. Nur wenn dieser Schalter

(mit `$B+` oder via *Options/Compiler*) aktiviert wird, berechnet das Programm zusammengesetzte logische Ausdrücke vollständig.

Reihenfolge der Berechnung

Wie durch den ANSI-Standard festgelegt, können Teile zusammengesetzter Ausdrücke unabhängig von der Reihenfolge ihrer Angabe berechnet werden. Wenn wir davon ausgehen, daß *F* und *G* Funktionen mit dem Ergebnistyp *Integer* sind, dann codiert der Compiler in einer Zuweisung wie

```
I := F(J) div G(J);
```

den Aufruf der Funktion *G* zuerst, weil dadurch die abschließende Division einfacher auszuführen ist. Aus diesem Grund schreibt die Sprachdefinition von Pascal vor, daß derartige Ausdrücke nicht mit Seiteneffekten arbeiten dürfen: wenn der Wert von *J* im oben gezeigten Beispiel durch den Aufruf einer der beiden Funktion verändert wird, hätte *I* sonst ein Ergebnis, das von der Arbeitsweise des Compilers abhängig wäre.

Falls es notwendig sein sollte, mit Seiteneffekten zu arbeiten, müssen Sie eine temporäre Variable und voneinander getrennte Funktionsaufrufe verwenden, also:

```
Temp := F(J);  I := Temp div G(J);
```

Hinweis: Das Kurzschlußverfahren bei der Bewertung boolescher Ausdrücke stellt eine Ausnahme von dieser Regel dar - nach der Aktivierung mit `{$B-}` werden über **and** und **or** verbundene Operanden *immer* von links nach rechts ausgewertet.

Überprüfung von Wertebereichen

Indizierungen, Konstanten-Zuweisungen an Variablen sowie Prozedur- und Funktionsaufrufe mit Konstanten als Parameter werden während der Compilierung daraufhin geprüft, ob sie innerhalb der zulässigen Wertebereiche liegen – dadurch wird der entsprechende Code zur Laufzeit des Programms eingespart. Das folgende Programm enthält drei Fehler, die alle bereits vom Compiler entdeckt werden:

```
var
  X: Byte;
  AX: array[1..10] of Byte;
procedure Dummy (S: ShortInt);
```

```
begin
   ...
end;
```

```
begin
  AX[12] := 100;   { unzulässige Indizierung }
  X := 999;      { Zuweisung außerhalb des Wertebereichs }
  Dummy(200);    { Parameter außerhalb des Wertebereichs }
end.
```

Schiebebefehle anstelle von Multiplikationen

Integer-Multiplikationen der Form X * C werden über den Prozessorbefehl SHL codiert, wenn C eine Konstante mit dem Wert einer Zweierpotenz ist.

In ähnlicher Weise geschieht die Indizierung von Arrays: wenn die Größe eines Array-Elements als Zweierpotenz darstellbar ist (2, 4, 8 ... 256 Bytes), dann verwendet Turbo Pascal zur Berechnung der absoluten Adresse den Befehl SHL anstelle einer Multiplikation.

Wortweise Ausrichtung von Daten

Mit der Standardvorgabe {$A+} sorgt der Compiler dafür, daß sämtliche Variablen außer *Char*, *Byte* und *ShortInt* auf geradzahligen Adressen im Speicher beginnen, und macht so Zugriffe des Prozessors 8086 und seiner Nachfolger schneller. Auf Maschinen mit einem 8088, die nur über einen Datenbus mit 8 Bit Breite verfügen, hat diese Optimierung keine Wirkung.

Der Compiler-Befehl {$A} steuert die Ausrichtung: Im Modus {$A+} werden Daten wie oben beschrieben ausgerichtet, im Modus {$A-} unterbleibt die Ausrichtung.

Entfernung »toten« Codes

Befehle und Anweisungen, die aus logischen Gründen nie ausgeführt werden können, entfernt der Compiler selbständig. Zwei Beispiele:

```
const
  MonoCRT = False;
...
 if MonoCRT then ...
```

```
while false do ...
```

Hier wird weder für die **if**-Bedingung noch für die **while**-Schleife Code erzeugt.

Intelligentes Linken

Der in Turbo Pascal eingebaute Linker nimmt nur die Funktionen und Prozeduren in die .EXE-Datei auf, die auch tatsächlich verwendet werden – Routinen, für die das Programm keine Aufrufe enthält, werden automatisch entfernt.

Analoges gilt für Datendeklarationen, auf die das Programm nicht zugreift. Das »Aussortieren« unnötigen Codes arbeitet auf der Basis einzelner Routinen; das Entfernen von Daten dagegen auf der Basis einzelner Deklarations-Abschnitte. Ein Beispiel zur Verdeutlichung:

```
program SmartLink;

const
  H: array[0..15] of Char = '0123456789ABCDEF';

var
  I,J: Integer;
  X,Y: Real;

var
  S: string[79];

var
  A: array[1..10000] of Integer;

procedure P1;
begin
  A[1] := 1;
end;

procedure P2;
begin
 I:= 1;
end;

procedure P3;
begin
  S:= 'Turbo Pascal';
  P2;
end;

begin
  P3;
end.
```

Das Hauptprogramm ruft hier die Prozedur *P3* auf, die ihrerseits einen Aufruf von *P2* enthält – folglich werden diese beiden Routinen in die .EXE-Datei aufgenommen. *P2* greift mit der Zuweisung an *I* auf die erste **var**-Sektion des Programms zu, *P3* mit der Zuweisung an *S* auf die zweite – die .EXE-Datei enthält damit die Variablen I, J, X, Y und S.

Die Prozedur *P1* wird an keiner Stelle verwendet und deshalb vom Linker aus der .EXE-Datei hinausgeworfen; dasselbe gilt für die Variable *A* (auf die nur innerhalb der entfernten Routine *P1* zugegriffen wird) und die typisierte Konstante *H*, auf die überhaupt kein Bezug des Programms existiert.

Dieses »intelligente« Verhalten des Linkers macht sich vor allem bei der Aufnahme von Units bemerkbar, die mehr oder weniger als Bibliotheken aufgebaut sind: Die Unit *WinDos* deklariert beispielsweise eine recht hohe Zahl von Prozeduren, Funktionen und Datenstrukturen, die praktisch nie in einem Programm auf einmal verwendet werden. Jedes Programm, das *WinDos* verwendet, bekommt exakt die Dinge zugewiesen, die tatsächlich notwendig sind – der restliche Code wird zusammen mit überflüssigen Datenstrukturen entfernt.

21. Compiler-Befehle

Compiler-Befehle weisen den Compiler an, bestimmte Operationen auszuführen oder zu unterlassen. Ein Compiler-Befehl ist ein Kommentar mit einer speziellen Syntax. Turbo Pascal läßt Compiler-Befehle überall da zu, wo auch Kommentare zugelassen sind.

Das erste Zeichen nach dem öffnenden Kommentarbegrenzer eines Compiler-Befehls ist das Dollarzeichen ($). Darauf folgt eine Name (ein oder mehr Buchstaben), der den jeweiligen Befehl bezeichnet. Compiler-Befehle lassen sich in drei Klassen einteilen: Schalter, Parameter und Bedingungen.

- Mit einem *Schalter* werden bestimmte Funktionen des Compilers ein- oder ausgeschaltet. Dementsprechend ist die Syntax: auf den Namen des Befehls folgt entweder ein Minuszeichen (-) oder ein Pluszeichen (+). Für jeden Schalter existiert eine Standardvorgabe; ein einmal gesetzter Schalter bleibt solange aktiv, bis er durch einen erneuten Compiler-Befehl wieder zurückgesetzt wird (oder bis zum Ende der Compilierung).

- Ein *Parameter* legt Werte (wie die Größe des Stacks) und Dateinamen fest. Auch hier wird in den meisten Fällen mit Standardvorgaben gearbeitet, wenn Sie keine eigenen Angaben machen. Im Gegensatz zu Schaltern muß immer ein Leerzeichen zwischen dem Befehlsnamen und dem folgenden Parameter stehen.

- Eine *Bedingung* kontrolliert, welche Teile eines Quelltextes compiliert werden (»bedingte Compilierung«). Für Compiler-Befehle dieser Art gibt es keine Standardvorgaben: normalerweise bearbeitet der Compiler den gesamten Inhalt einer Quelltext-Datei. Auch hier muß zwischen dem Befehl und der Bedingung mindestens ein Leerzeichen stehen.

Alle Befehle außer den Schaltern müssen zumindest ein Leerzeichen zwischen Befehlsname und den Parametern haben. Im folgenden finden Sie einige Beispiele für Compiler-Befehle:

```
{$B+}                              { vollständige Auswertung
                                     boolscher Ausdrücke }
{$R-  Bereichsüberprüfung aus }    { Kommentar im Befehl! }
{$I D_TYPEN.INC}                   { Include-Datei D_TYPEN.INC }
{$DEFINE Debug}                    { Definition des Symbols Debug }
{$IFDEF Debug}                     { Wenn Debug definiert ist, dann }
```

```
  ...                              { wird dieser Teil compiliert }
  ...                              { - ansonsten nicht }
{$ENDIF}                           { Ende der bedingten Compilierung }
```

Compiler-Befehle können direkt in den Quelltext eingefügt werden. Außerdem können die Standardbefehle für den Kommandozeilencompiler (TPCW.EXE) und die IDE (TPW.EXE) geändert werden. Im Dialogfenster *Options/Compiler* können Sie die meisten der Compiler-Befehle einstellen. Beachten Sie, daß Veränderungen an den Einstellungen alle nachfolgenden Compilierungen betreffen. Wenn Sie die Kommandozeilen-Version des Compilers verwenden, können Sie Compiler-Befehle in der Kommandozeile angeben (zum Beispiel TPCW /$R+ MEINPROG), oder Compiler-Befehle in die Konfigurationsdatei einfügen (Weitere Informationen hierzu finden Sie in Kapitel 8 »Die Kommandozeilen-Version des Compilers« des Benutzerhandbuchs). Compiler-Befehle im Quelltext setzen sowohl die Standardwerte im Kommandozeilencompiler als auch in der IDE außer Kraft.

Wenn Sie in der IDE arbeiten, den alternativen Befehlssatz des Editors verwenden und nach einen schnellen Möglichkeit suchen, um die Auswirkungen bestimmter Compiler-Befehle zu sehen, drücken Sie einfach `[Ctrl][O][O]`. Turbo Pascal fügt dann die aktuellen Einstellungen im oberen Teil des Edit-Fensters ein.

Schalter

Compiler-Schalter arbeiten entweder *global* (für die gesamte Compilierung) oder *lokal* (bis zu einer erneuten Umschaltung). Globale Schalter müssen grundsätzlich nach dem einleitenden **program** oder **unit** und vor jedem anderen Teil des Programms stehen (d.h. vor dem ersten **uses**, **label**, **const**, **type**, **procedure**, **function** oder **begin**). Lokale Schalter können dagegen an beliebiger Stelle des Programms erscheinen.

Turbo Pascal erlaubt es, mehrere Schalter in einer Kommentarklammer zusammenzufassen, solange sie durch Kommas voneinander getrennt sind. Das Dollarzeichen erscheint in diesem Fall nur einmal, Leerzeichen oder »echte« Kommentare *zwischen* den Befehlen sind nicht erlaubt. Zum Beispiel:

```
{$B+,R-,S-}
```

Ausrichtung von Variablen im Speicher

Syntax:	{$A+} oder {$A-}
Standard:	{$A+}
Typ:	global
Menü-Äquivalent:	*Options/Compiler/Word align data*

Dieser Schalter legt fest, wie Compiler und Linker Variablen mit einem Platzbedarf von mehr als einem Byte im Speicher anordnen.

Im Modus {$A+} (»Word«) werden alle Variablen außer *Char* und *Byte* so im Speicher angeordnet, daß der durch sie belegte Bereich mit einer geradzahligen

Adresse beginnt. Zugriffe des 8086 und seiner Nachfolger werden dadurch beschleunigt; bei einem 8088, der nur jeweils 8 Bit auf einmal lesen kann, hat die Anordnung der Variablen keine weiteren Wirkungen.

Im Modus {$A-} (»Byte«) findet keine spezielle Ausrichtung statt. Variablen werden hier so dicht wie möglich hintereinander gepackt.

Hinweis: **var-** und **const**-Deklarationsabschnitte werden unabhängig vom Stand des Schalters {$A} vom Linker grundsätzlich mit einer geraden Speicheradresse begonnen. Analoges gilt für den Stackzeiger (Prozessor-Register SP), der immer eine gerade Adresse enthält: beim Belegen von lokalem Speicherplatz fügt der Compiler nötigenfalls ein »Leerbyte« hinzu.

Auswertung boolscher Ausdrücke

Syntax:	{$B+} oder {$B-}
Standard:	{$B-}
Typ:	lokal
Menü-Äquivalent:	*Options/Compiler/Complete boolean evaluation*

Dieser Schalter legt fest, wie mit **and** und **or** verbundene boolsche Ausdrücke ausgewertet werden.

Im Modus {$B+} (»Complete«) erzeugt der Compiler Code für eine vollständige Auswertung sämtliche Teile eines zusammengesetzten Ausdrucks werden auch dann berechnet, wenn das Ergebnis des Gesamtausdrucks bereits feststeht.

Im Modus {$B-} (»Short Circuit«) wird ein *Kurzschlußverfahren* benutzt: die Teile eines Ausdrucks werden grundsätzlich in der Reihenfolge ihrer Angabe ausgewertet; die Auswertung bricht ab, sobald sich am Gesamtergebnis nichts mehr ändern kann (für **or**: ein Teilausdruck ergibt *True*, für **and**: ein Teilausdruck ergibt *False*). Beispiele für die Unterschiede zwischen beiden Verfahren finden Sie in Kapitel 6 »Ausdrücke«.

Zusatzinformationen zur Fehlersuche

Syntax:	{$D+} oder {$D-}
Standard:	{$D+}
Typ:	global
Menü-Äquivalent:	*Options/Compiler/Debug information*

Dieser Schalter legt fest, ob der Compiler zusätzliche Informationen zur Fehlersuche erzeugt oder nicht.

Zusatzinformationen zur Fehlersuche bestehen aus einer Tabelle mit Zeilennummern für jede Routine, über die die Verbindung zwischen den Adressen der einzelnen Befehle und dem Quelltext hergestellt werden kann.

Bei der Compilierung von Units werden die Zusatzinformationen zusammen mit Code und Daten direkt in der entsprechenden .TPU-Datei gespeichert. Die Compilierung im Modus {$D+} vergrößert .TPU- und .EXE-Dateien bzw. vergrößert den Platzbedarf bei einer Compilierung im Hauptspeicher Größe und Arbeitsgeschwindigkeit eines fertigen Programms bleiben aber unbeeinflußt.

$D hat keinen Auswirkungen, solange Sie das Auswahlfeld *Debug info in EXE* im Dialogfenster *Options/Linker* in der IDE nicht angewählt haben oder bei der

Verwendung von TPCW.EXE den Parameter /V in der Kommandozeile angegeben haben.

Wenn die Option *Debug Info* für ein bestimmtes Programm oder eine Unit angewählt wurde, können Sie Turbo Debugger für Windows verwenden, um dieses Modul im Einzelschritt-Modus ablaufen zu lassen oder Breakpoints zu setzen.

Ist eine Option des Schaltfelds *Map file* im Dialogfenster *Options/Linker* erzeugt der Compiler nur dann vollständige Informationen zu einem bestimmten Modul, wenn dieses Modul im {$D+}-Modus compiliert wurde.

Der Schalter {$D+} wird meist in Verbindung mit dem Schalter {$L} verwendet, um festzulegen, ob Daten über lokale Symbole zusammen mit den Zeilennummern gespeichert werden oder nicht.

Erzwingen von *far*-Aufrufen

Syntax:	{$F+} oder {$F-}
Standard:	{$F-}
Typ:	lokal
Menü-Äquivalent:	*Options/Compiler/Force far calls*

Dieser Schalter legt fest, ob der Compiler bei nachfolgenden Prozeduren und Funktionen selbst über die Art des Aufrufs entscheidet oder ob er Aufrufe und Rücksprünge grundsätzlich als **far** codiert. Im Modus {$F+} wird (außer für lokale Prozeduren und Funktionen) immer mit **far**-Aufrufen und entsprechenden Rücksprungbefehlen gearbeitet. Im Modus {$F-} werden Prozeduren und Funktionen nur dann als **far** aufgerufen, wenn sie im Interface-Teil einer Unit erscheinen und damit auch anderen Modulen zur Verfügung stehen, ansonsten geschieht die Codierung als **near**.

Weitere Informationen zu **near**- und **far**-Aufrufen finden Sie in Kapitel 18 »Interne Details«.

Erzeugung von *80286*-Code

Syntax:	{$G+} oder {$G-}
Standard:	{$G-}
Typ:	lokal
Menü-Äquivalent:	*Options/Compiler/286 instructions*

Dieser Schalter legt fest, ob der Compiler Code für den 80286-Prozessor erzeugt oder nicht. Im Modus {$G-} werden ausschließlich 8086-Befehle erzeugt. Solchermaßen compilierte Programme sind auf allen Rechnern mit Prozessoren der 80x86-Familie lauffähig.

Im Modus {$G+} hingegen verwendet der Compiler die zusätzlichen Befehle des 80286-Prozessors zur Erhöhung der Effizienz des Codes. Folgende 80286-Befehle sind im Modus {$G+} verfügbar: ENTER, LEAVE, PUSH*konstante*, erweitertes IMUL sowie die erweiterten SHL und SHR. Programme, die im Modus {$G+} compiliert wurden, sind auf Rechnern mit 8088- bzw. 8086-Prozessoren nicht mehr lauffähig.

Automatische Prüfung von Ein-/Ausgaben (»I/O«)
Syntax: {$I+} oder {$I-}
Standard: {$I+}
Typ: lokal
Menü-Äquivalent: *Options/Compiler/I/O checking*

Dieser Schalter legt fest, ob Ein- und Ausgaben zur Laufzeit des Programms automatisch auf Fehler überprüft werden.

Im Modus {$I+} erzeugt der Compiler nach jedem Ein-/Ausgabebefehl automatisch zusätzlichen Prüfcode, der das Programm im Falle eines Fehlers mit einer entsprechenden Meldung abbricht. Im Modus {$I-} muß die Fehlerprüfung durch das Programm *über* Aufrufe der Funktion *IOResult* erfolgen.

/$L – Lokale Symbole
Syntax: {$L+} oder {$L-} bzw. /$L+ oder /$L-
Standard: {$L+}
Typ: global
Menü-Äquivalent: *Options/Compiler/Local symbols*

Dieser Schalter legt fest, ob Turbo Pascal neben den über {$D} (»Debug information«) erzeugten Zeilennummern zusätzlich Informationen über »lokale« Symbole in ein Modul aufnimmt oder nicht.

Im Modus {$L+} berücksichtigt der Compiler nicht nur die Namen und Datentypen aller lokalen Variablen und Konstanten des jeweiligen Moduls bei der Compilierung von Units werden auch die Routinen (zusammen mit ihren lokalen Symbolen) verzeichnet, die nur im Implementationsteil deklariert sind.

Hinweis: Bei einer Compilierung im Modus {$D-} also ohne das Erzeugen von Informationen zur Fehlersuche bleibt der Schalter {$L} wirkungslos.

$L hat keinen Auswirkungen, solange Sie das Auswahlfeld *Debug info in EXE* im Dialogfenster *Option/Linker* der IDE nicht angewählt oder bei der Verwendung von TPCW.EXE den Parameter /V in der Kommandozeile nicht angegeben haben.

Wenn lokale Symbole für ein bestimmtes Programm oder eine Unit generiert wurden, können Sie Turbo Debugger für Windows verwenden, um die lokalen Variablen unter Verwendung der Symbolnamen in diesem Modul zu überprüfen und zu verändern. Dadurch wird die Fehlersuche und -behebung wesentlich vereinfacht. Weitergehend können Aufrufe der Prozeduren und Funktionen des Moduls über das Fenster *View/Stack* überprüft werden.

Die *Map file*-Optionen im Dialogfenster *Options/Linker* und die *Debug info*??-Optionen erzeugen nur dann Informationen zu lokalen Symbolen, wenn dieses Modul im {$D+}-Modus compiliert wurde.

Der Schalter {$L} wird meist mit dem Schalter {$D} verwendet. Dadurch wird die Erzeugung von Zeilennummerntabellen zur Fehlersuche und -behebung ein- bzw. ausgeschaltet. Beachten Sie, daß der Befehl $L ignoriert wird, wenn sich der Compiler im Modus {$D-} befindet.

Bereichs-Überprüfung

Syntax:	{$R+} oder {$R-}
Standard:	{$R-}
Typ:	lokal
Menü-Äquivalent:	*Options/Compiler/Range checking*

Dieser Schalter legt fest, ob der Compiler bei der Indizierung von Strings und Arrays sowie bei Zuordnungen zu skalaren Typen und Unterbereichen zusätzlichen Prüfcode erzeugt.

Im Modus {$R+} wird vor jeder Indizierung und Zuordnung geprüft, ob sie sich innerhalb der zulässigen Grenzen bewegt falls nicht, bricht die Programmausführung mit einer entsprechenden Meldung ab.

Zusätzlich werden im Modus {$R+} sämtliche Aufrufe virtueller Methoden auf den Initialisierungs-Status der aufrufenden Objektinstanz hin überprüft. Wurde die aufrufende Instanz noch nicht durch ihren Konstruktor initialisiert, wird ebenfalls ein Laufzeitfehler ausgelöst.

Die zusätzliche Bereichs-Überprüfung vergrößert ebenso wie auch die Überprüfung der Aufrufe virtueller Methoden den Umfang der .EXE-Datei und setzt die Ausführungsgeschwindigkeit erheblich herab. Sie sollten den Schalter {$R+} deshalb ausschließlich in der Testphase aktivieren.

Stack-Prüfung

Syntax:	{$S+} oder {$S-}
Standard:	{$S+}
Typ:	lokal
Menü-Äquivalent:	*Options/Compiler/Stack checking*

Dieser Schalter legt fest, ob der Compiler zusätzlichen Prüfcode vor Belegungen des Stacks erzeugt.

Im Modus {$S+} wird vor jedem Prozedur- und Funktionsaufruf geprüft, ob noch genügend Platz zur Speicherung von Rücksprungadressen, lokalen Variablen usw. auf dem Stack vorhanden ist falls nicht, bricht das Programm mit einer entsprechenden Meldung ab. Auch wenn diese Prüfung den Umfang des Programms erhöht und die Ausführungsgeschwindigkeit herabsetzt, sollte sie wirklich erst dann abgeschaltet werden, wenn das Programm völlig fehlerfrei ist: Stack-Überläufe haben normalerweise einen Systemabsturz zur Folge und sind äußerst schwer zu lokalisieren.

Überprüfung von var-Strings

Syntax:	{$V+} oder {$V-}
Standard:	{$V+}
Typ:	lokal
Menü-Äquivalent:	*Options/Compiler/strict var-strings*

Dieser Schalter legt fest, welche Prüfungen der Compiler bei der Übergabe von Strings als **var**-Parameter vornimmt.

Im Modus {$V+} (»Strict«) setzt der Compiler voraus, daß formale und aktuelle String-Parameter einen *identischen* Typ haben, und weist jede Verletzung dieser Regel als Fehler aus.

Im Modus {$V-} (»Relaxed«) kann jeder String als **var**-Parameter übergeben werden selbst dann, wenn die Länge des formalen und die des aktuellen Strings nicht gleich sind.

Windows-Stackrahmen

Syntax:	{$W+} oder {$W-}
Standard:	{$W+}
Typ:	lokal
Menü-Äquivalent:	*Options/Compiler/Windows Stack Frames*

Dieser Schalter steuert die Erzeugung von Windows-spezifischem Prozedur-Eintritts- und Austrittscode für **far**-Prozeduren und -Funktionen. Im Modus {$W+} wird ein spezieller Eintritts- und Austrittscode für **far**-Prozeduren erzeugt, so daß der Windows-Speichermanager im Real-Modus **far**-Stackbereiche korrekt identifizieren kann, wenn er Veränderungen an der Call-Kette macht, nachdem ein Code- oder Datensegment bewegt wurde. Im Modus {$W-} wird dieser zusätzliche Eintritts- und Austrittscode nicht erzeugt. $W kann ausgeschaltet werden, wenn ein Programm nur im Standard- oder erweiterten 386er-Modus laufen soll.

Erweiterte Syntax

Syntax:	{$X+} oder {$X-}
Standard:	{$X-}
Typ:	global
Menü-Äquivalent:	*Options/Compile/Extended Syntax*

Dieser Schalter legt fest, ob die erweiterte Syntax von Turbo Pascal eingesetzt werden kann oder nicht.

■ Funktionsanweisungen. Im Modus {$X+} ist es möglich, Funktionsaufrufe als Anweisungen zu verwenden, d.h. eine Funktion aufzurufen, ihr Ergebnis jedoch zu verwerfen. Eigentlich zeichnet sich eine Funktion gerade dadurch aus, daß ihre Berechnungen immer zu einem Ergebnis führen, und daher erscheint es nur wenig sinnvoll, ein Funktionsergebnis zu verwerfen. In manchen Situationen ist bei einem Funktionsaufruf jedoch nicht das Funktionsergebnis relevant, sondern vielmehr die Ausführung verschiedener Operationen in Abhängigkeit der Funktionsparameter. Der Schalter {$X+} ermöglicht für solche Fälle die Handhabung einer Funktion entsprechend einer Prozedur.

Hinweis: Der Schalter {X+} hat keine Auswirkung auf die Handhabung der Funktionen der Unit *System*.

Null-terminierte Strings. Der Compiler-Befehl {$X+} aktiviert die Unterstützung null-terminierter Strings in Turbo Pascal. Dazu werden spezielle Regeln verwendet, die für Zeichenarrays (deren Index mit Null beginnt) im allgemeinen und für den vordefinierten Typ *PChar* gelten. Weitere Informationen dazu finden Sie in Kapitel 13 »Die Unit Strings«.

Parameter

Parameter-Befehle ermöglichen die Angabe von Parametern, die die Art und Weise der Compilierung beeinflussen.

Code-Segmentattribute

Syntax:	{$C *Attribut Attribut*}
Standard:	{$C MOVEABLE PRELOAD PERMANENT}
Typ:	global

Dieser Befehl wird verwendet, um die Attribute eines Code-Segments zu bestimmen. Jedem Code-Segment in einem Programm oder einer Bibliothek kann eine Reihe von Attributen zugeordnet werden, die das Verhalten des Code-Segments bestimmen, wenn es in den Speicher geladen wird. Zum Beispiel können Sie ein Code-Segment als *moveable* (beweglich) deklarieren. Das bedeutet, daß Windows dieses Code-Segment je nach Bedarf im Speicher bewegen kann. Oder das Code-Segement wird als *fixed* (gebunden) definiert, wodurch seine Position im Speicher nicht verändert werden kann.

Der Befehl {$C} betrifft nur das Code-Segment des Modules (Unit, Programm oder Bibliothek), in dem es angegeben wurde. In der folgenden Tabelle finden Sie die Code-Segment-Attributoptionen in Zweiergruppen geordnet; ein Mitglied einer Gruppe schließt das jeweils andere aus.

Moveable	Windows kann die Position des Code-Segments im Speicher verändern.
Fixed	Windows kann die Position des Code-Segments im Speicher nicht verändern.
Preload	Das Code-Segment wird geladen, wenn das Programm geladen wird.
DemandLoad	Das Code-Segment wird geladen, wenn es benötigt wird.
Permanent	Das Code-Segment verbleibt nach dem Laden im Speicher.
Discardable	Das Code-Segment kann aus dem Speicher entfernt werden, wenn es nicht mehr benötigt wird.

Als Voreinstellung gilt die erste Option jeder Gruppe. Mit dem Befehl {$C} können mehrere Code-Segment-Attribute angegeben werden. Wenn beide Optionen einer Gruppe im Befehl $C angegeben sind, ist nur die letzte aktiv. Im folgenden Beispiel wird das Code-Segment als beweglich definiert und kann aus dem Speicher entfernt werden, wenn es nicht länger benötigt wird.

```
{$C Fixed Moveable Discardable}
```

Programm-Beschreibung

Syntax:	{$D *text*}
Typ:	global

Dieser Befehl fügt den angegebenen Text in eine .EXE-Datei oder eine DLL ein. Der Text besteht üblicherweise aus dem Copyright-Hinweis und/oder der Versionsnummer. Es kann aber auch beliebiger anderer Text eingefügt werden.

Dazu ein Beispiel:

```
{$D XL-5 Version 12.5 (C) MOL D&C, 1991}D
```

Include-Datei
Syntax: {$I *Dateiname*}
Typ: lokal

Dieser Befehl weist den Compiler an, die durch *Dateiname* bezeichnete Datei einzulesen. Der Compiler verhält sich dabei im Prinzip so, als ob der gelesene Text anstelle des {$I}-Befehls stünde die eingelesene Datei muß allerdings einen abgeschlossenen Programmblock enthalten (d.h. vollständige Prozeduren, Funktionen und/oder ein komplettes Hauptprogramm inklusive **begin** und **end**).

Wenn *Dateiname* kein Suffix enthält, wird automatisch .PAS angenommen. Wenn *Dateiname* keinen Suchweg angibt, dann sucht Turbo Pascal zuerst im momentan gesetzten Directory und danach in sämtlichen Directories, die via *Options/Directories/Include directories* bzw. durch den TPCW-Kommandozeilen-Parameter */I* angegeben sind.

Turbo Pascal erlaubt die Verschachtelung von Include-Dateien bis zu 15 Ebenen.

Bei der Verwendung von Include-Dateien gibt es jedoch einen Einschränkung. Eine Include-Datei kann nicht in der Mitte eines Anweisungsteils angegeben werden. Alle Anweisungen zwischen **begin** und **end** im Anweisungsteils müssen in derselben Quelldatei vorliegen.

Einbinden von Object-Dateien
Syntax: {$L *Dateiname*}
Typ: lokal

Dieser Befehl weist den Linker an, die durch *Dateiname* bezeichnete Datei mit in das Programm aufzunehmen. Die Datei muß im Intel-Object-Format vorliegen (also mit einem Assembler erzeugt sein); wenn kein Suffix angegeben ist, hängt Turbo Pascal automatisch die Endung .OBJ an *Dateiname* an.

Bezeichner der Object-Datei, auf die von Pascal aus zugegriffen wird, müssen innerhalb der Object-Datei als PUBLIC und innerhalb des Pascal-Quelltextes als **external** deklariert sein.

Wenn *Dateiname* keinen Suchweg angibt, dann sucht Turbo Pascal zuerst im momentan gesetzten Directory, und danach in sämtlichen Directories, die über *Options/Directories/Object directories* bzw. den TPCW-Kommandozeilen-Parameter /O angegeben wurden. Weitere Informationen hierzu finden Sie im Kapitel 23 »Einbinden von Assembler-Code«.

Speicherbelegung
Syntax: {$M *Stackgröße*, *Heapgröße*}
Standard: {$M 8192, 8192}
Typ: global
Menü-Äquivalent: *Options/Memory sizes*

Diese Parameter legen die Speicherbelegung eines Programms oder einer Bibliothek fest.

Zulässige Werte für die Stackgröße liegen im Bereich von 1024 bis 65520. Die Stackgröße wird in einer Bibliothek ignoriert (sie verwendet den Stack des Programms, das von ihr Gebrauch macht). Die Heapgröße muß im Bereich von 0..655360 liegen, womit die Größe des lokalen Heapbereichs im Datensegment angegeben wird.

Hinweis: Der Befehl {$M} hat bei der Compilierung einer Unit keine Wirkung.

Numerische Operationen

Syntax:	{$N+} oder {$N-}
Standard:	{$N-}
Typ:	global
Menü-Äquivalent:	*Options/Compiler/80x87 code*

Dieser Schalter legt fest, ob Operationen mit Realzahlen durch Routinen der Laufzeitbibliothek oder über die direkte Ansteuerung eines numerischen Coprozessors stattfinden.

Im Modus {$N-} (»Software«) steht nur der Datentyp *Real* zur Verfügung, alle Operationen mit diesem Typ geschehen über Aufrufe der Laufzeitbibliothek.

Im Modus {$N+} erzeugt Turbo Pascal Code, um die Rechenoperationen mit Realzahlen mit dem numerischen Coprozessor 8087 durchzuführen.

Ressourcen-Datei

Syntax:	{$R *Dateiname*}
Typ:	lokal

Dieser Befehl gibt den Namen der Ressourcen-Datei an, die in einer Anwendung oder einer Bibliothek eingebunden werden soll. Die Datei muß einen Windows-Ressourcen-Datei sein, die voreingestellte Dateinamenserweiterung ist .RES.

Ist der Befehl {$R *Dateiname*} in einer Unit angegeben, wird dieser Dateiname einfach in der .TPU-Datei vermerkt. Dabei findet keine Überprüfung statt, die sicherstellt, daß der Dateiname korrekt ist und es auch eine solche Datei gibt.

Während des Linkens der Anwendung oder Bibliothek (nachdem die Quelldatei des Programms oder der Bibliothek compiliert wurde), werden die Ressourcen-Dateien, die in den verwendeten Units, im Programm und in der Bibliothek angegeben sind, abgearbeitet. Jede Ressource in jeder Ressourcen-Datei wird in die enstehende .EXE-Datei oder DLL kopiert. Während der Ressourcen-Abarbeitung sucht der Linker im aktuellen Verzeichnis oder in den im Dialogfenster *Options/Directories/Resource Directories* (oder in den Verzeichnissen, die mit dem TPCW-Kommandozeilenbefehl /R angegeben wurden) nach .RES-Dateien.

Bedingte Compilierung

Turbo Pascal stellt zwei Konstrukte (ähnlich der Pascal-Anweisung **if**) zur Verfügung, mit denen Teile eines Quelltextes von der Compilierung ausgeschlossen bzw. in die Compilierung einbezogen werden können. Das erste Konstrukt hat die Form

```
{$IF xxxx}  ...  {$ENDIF}
```

Der mit {$IF} und {$ENDIF} eingeschlossene Text wird nur dann vom Compiler bearbeitet, wenn die mit *xxxx* angegebene Bedingung zutrifft und ansonsten ignoriert.

Das zweite Konstrukt stellt eine Erweiterung des ersten dar und hat die Form

```
{$IF xxxx}  ...  {$ELSE}  ...  {$ENDIF}
```

Wenn die durch *xxxx* angegebene Bedingung zutrifft, wird der mit {$IF} und {$ELSE} eingeschlossene Text vom Compiler bearbeitet (und der zwischen {$ELSE} und {$ENDIF} befindliche Text ignoriert). Trifft die durch *xxxx* angegebene Bedingung nicht zu, dann ignoriert der Compiler den Text zwischen {$IF}/{$ELSE} und bearbeitet den durch {$ELSE} und {$ENDIF} eingeschlossenen Teil.

Mit diesen beiden Befehlen zur bedingten Compilierung lassen sich Programme schreiben, die durch die Veränderung eines einzigen Symbols an verschiedene Gerätekonfigurationen angepaßt oder für verschiedene Zwecke verwendet werden können. Einige Beispiele dazu:

```
{$IFDEF Debug}
  Writeln('X = ', X);
{$ENDIF}

{$IFDEF CPU87}
  {$N+}
  type
    Real = Double;
{$ELSE}
  {$N-}
  type
    Single = Real;
    Double = Real;
    Extended = Real;
    Comp = Real;
{$ENDIF}
```

Befehle zur bedingten Compilierung lassen sich bis zu einer Tiefe von 16 Ebenen verschachteln. Über bedingte Befehle lassen sich nicht nur Typen definieren, Compiler-Schalter setzen oder Units aufnehmen auch Include-Dateien können darüber einbezogen oder ausgeschlossen werden. Für Include-Dateien gibt es allerdings eine Voraussetzung: Jede mit {$IF...} begonnene

Verschachtelungsebene muß innerhalb derselben Datei wieder abschlossen sein, d.h. jede Datei muß genausoviele {$IF...}- wie {$ENDIF}-Befehle enthalten.

Symbole zur bedingten Compilierung

Die Auswertung einer Bedingung *xxxx* für den Befehl {$IFDEF} ist recht einfach und im Stil boolscher Variablen gehalten: *xxxx* steht in jedem Fall für ein *Symbol*, das entweder definiert ist (in diesem Fall trifft die Bedingung zu) oder nicht (die {$IF...}-Bedingung ergibt *False*). Zur Definition und Löschung (»Undefinition«) derartiger Symbole gibt es zwei weitere Compiler-Befehle:

```
{$DEFINE Symbolname}     { definiert ein Symbol }
{$UNDEF Symbolname}      { löscht ein Symbol }
```

Sie können entweder /D im Kommandozeilencompiler verwenden, oder diese Angabe im Dialogfenster *Conditional Defines* über die Menüoption *Options/Compiler* aus der IDE machen.

Für ihre Namen gelten dieselben Regeln wie für Bezeichner von Turbo Pascal: das erste Zeichen muß ein Buchstabe sein, darauf können beliebige Kombinationen von Buchstaben, Ziffern und Unterstrichen (_) folgen. Zwischen Groß- und Kleinbuchstaben wird nicht unterschieden, die deutschen Umlaute und das »ß« sind nicht zulässig. Die Länge eines Namens ist beliebig, die ersten 63 Zeichen sind signifikant.

Hinweis: Symbole zur bedingten Compilierung und Pascal-Bezeichner sind vollständig voneinander getrennt. Dadurch können Probleme mit doppelt vergebenen Namen erst gar nicht entstehen, ein Zugriff auf bedingte Symbole von einem Pascal-Programm oder umgekehrt ist allerdings auch nicht möglich. Im folgenden Beispiel erhalten Sie weder die Fehlermeldung »doppelter Bezeichner« noch wird der *Writeln*-Befehl vom Compiler bearbeitet:

```
{$UNDEF Debug}   { Symbol zur bedingten Compilierung }

const
  Debug = True;   { Pascal-Variable }
begin
  {$IFDEF Debug}
    Writeln('Debug ist definiert...');
  {$ENDIF}
end.
```

Aus demselben Grund endet die Compilierung des nächsten Beispiels mit der Meldung »unbekannter Bezeichner«:

```
var              { diese »var-Sektion« ist erlaubt - sie hat aber }
{$DEFINE Debug}  { nichts mit einer Pascal-Variablen zu tun! }
begin
   if Debug then Writeln('Debug ist definiert...');
end.
```

Die folgende Liste gibt die Symbole wieder, die Turbo Pascal zu Beginn jeder Compilierung definiert. Diese Symbole können zusammen mit {$IF} verwendet (oder auch mit {$UNDEF} gelöscht) werden.

VER10: steht für die Version **1.0** von Turbo Pascal für Windows; folgende Versionen werden ebenfalls eigene Symbole definieren (VER20 für die Version 2.0 usw).

WINDOWS: steht dafür, daß der Compiler das Betriebssystem MS-WINDOWS voraussetzt. Versionen von Turbo Pascal, die unter anderen Betriebssystemen arbeiten, werden ein entsprechendes Symbol anderen Namens definieren.

CPU86: steht dafür, daß der Compiler einen Prozessor der Familie 80x86 voraussetzt. Versionen von Turbo Pascal, die für andere Prozessorfamilien geschrieben sind, werden ein entsprechendes Symbol anderen Namens definieren.

CPU87: ist dann definiert, wenn zum Zeitpunkt der Compilierung ein numerischer Coprozessor (8087) ermittelt wurde. Über dieses Symbol läßt sich das Setzen des Schalters {$N} automatisieren:

```
{$IFDEF CPU87} {$N+} {$ELSE} {$N-} {$ENDIF}
```

Andere Symbole zur bedingten Compilierung können vor der Compilierung über das Dialogfenster *Conditional Defines* (Menüoption *Options/Compiler*) oder den TPC- Kommandozeilenbefehl /D definiert werden.

Der bedingte Compiler-Befehl DEFINE
Syntax: {$DEFINE *Symbolname*}

Definiert das durch *Symbolname* angegebene Symbol, das vom Zeitpunkt seiner Definition bis zum Ende der Compilierung oder einem entsprechenden {$UNDEF}-Befehl erhalten bleibt. Wenn *Symbolname* vor diesem Befehl bereits definiert war, dann hat {$DEFINE} keine Wirkung.

Der bedingte Compiler-Befehl UNDEF
Syntax: {$UNDEF *Symbolname*}

Löscht (»undefiniert«) das durch *Symbolname* bezeichnete Symbol bis zum Ende der Compilierung bzw. bis zu seiner erneuten Definition via {$DEFINE}. Wenn *Symbolname* vor diesem Befehl überhaupt nicht definiert war, dann hat {$UNDEF} keine Wirkung.

Der bedingte Compiler-Befehl IFDEF
Syntax: {$IFDEF *Symbolname*}

Bewirkt die Compilierung des folgenden Textes, wenn das durch *Symbolname* bezeichnete Symbol definiert ist. Ansonsten wird der folgende Text bis zum nächsten $ENDIF oder $ELSE ignoriert.

Der bedingte Compiler-Befehl IFNDEF
Syntax: {$IFNDEF *Symbolname*}

Das logische Gegenstück zu {$IFDEF} nachfolgender Text wird nur compiliert, wenn das durch *Symbolname* bezeichnete Symbol **nicht** definiert ist.

Der bedingte Compiler-Befehl IFOPT
Syntax: {$IFOPT *Compiler-Schalter*}

Bestimmt die Compilierung nachfolgenden Textes abhängig vom Stand eines angegebenen Compiler-Schalters. *Compiler-Schalter* steht hier für den Namen eines Schalters, gefolgt von einem Plus- oder Minuszeichen. Wenn die Angabe mit dem tatsächlichen Stand des Schalters übereinstimmt, gilt die Bedingung als erfüllt. Die **type**-Definition im folgenden Beispiel wird also nur dann compiliert, wenn der Schalter $N auf {$N+} gesetzt ist:

```
{$IFOPT N+}
  type  Real = Extended;
{$ENDIF}
```

Der bedingte Compiler-Befehl ELSE
Syntax: {$ELSE}

Leitet den **else**-Zweig eines mit {$IFDEF}, {$IFNDEF} oder {$IFOPT} begonnenen Konstrukts ein und bezieht sich immer auf das letzte noch nicht abgeschlossene {$IF...}.

Der bedingte Compiler-Befehl ENDIF
Syntax: {$ENDIF}

Schließt ein mit {$IFDEF}, {$IFNDEF} oder {$IFOPT} begonnenes Konstrukt ab und bezieht sich immer auf das letzte noch nicht abgeschlossene {$IF...}.

IV. Turbo Pascal und Assembler

22. Der integrierte Assembler

Der integrierte Assembler von Turbo Pascal erlaubt Ihnen, 8086/8087- bzw. 80286/80287-Assembler-Anweisungen direkt in den Pascal-Quelltext einzufügen. Künftig ist es somit nicht mehr nötig (obwohl immer noch möglich), per Hand Maschinencode für **inline**-Anweisungen zu erzeugen oder **external**-Prozeduren und -Funktionen in Ihr Pascal-Programm einzubinden.

Die Syntax des integrierten Assemblers entspricht weitgehend der Syntax von Turbo Assembler bzw. Microsofts Makro-Assembler. Sein Befehlssatz umfaßt sämtliche Befehle der 8086/8087- und 80286/80287-Prozessoren sowie die meisten Turbo-Assembler-Operatoren.

Mit Ausnahme der Anweisungen *DB*, *DW* und *DD*, unterstützt der integrierte Assembler keine weiteren Turbo-Assembler-Anweisungen (also z.B. weder *EQU*, *PROC*, *STRUC*, *SEGMENT* noch *MACRO*). Meist haben jedoch Operationen, die bei Turbo Assembler über Anweisungen implementiert sind, eine Entsprechung in Turbo-Pascal-Konstrukten. Die meisten *EQU*-Anweisungen können zum Beispiel durch Konstanten-, Variablen- oder Typdeklarationen gleicher Bedeutung ersetzt werden. Die *PROC*-Anweisung findet ihre Entsprechung in Prozedur- und Funktionsdeklarationen, genauso wie die *STRUC*-Anweisung den **record**-Typen von Turbo Pascal gleicht. Somit kann man sich den integrierten Assembler fast als einen Maschinensprache-Compiler vorstellen, der für alle Deklarationen die Pascal-Syntax benutzt.

Die Anweisung asm

Assembler-Blöcke werden durch eine **asm**-Anweisung eingeleitet und mit **end** abgeschlossen. Die Syntax von **asm**-Anweisungen lautet:

```
asm  AsmBefehl  Trennzeichen AsmBefehl   end
```

```
if EnableInts then
  asm
    STI
  end
  else
  asm
```

```
    CLI
  end

asm
  MOV AX,Links;  XCHG AX,Rechts;  MOV Links,AX;
end;

asm
  MOV   AX,0              (* Funktion zum Lesen der Tastatur *)
  INT   16H               (* BIOS-Aufruf *)
  MOV   CharCode,AL       (* ASCII-Code sichern *)
  MOV   ScanCode,AH       (* Scan-Code sichern  *)
end;

asm
  PUSH  DS                (* Daten-Segment sichern *)
  LDS   SI,Source         (* Quellzeiger laden *)
  LES   DI,Dest           (* Zielzeiger laden *)
  MOV   CX,Count          (* Blockgröße laden *)
  CLD                     (* Direction-Flag setzen *)
  REP   MOVSB             (* Block kopieren *)
  POP   DS                (* Daten-Segment restaurieren *)
end;
```

Wie Sie sehen, ist es durchaus zulässig, mehrere Assembler-Anweisungen in eine einzige Zeile zu schreiben. Beachten Sie jedoch, daß Assembler-Anweisungen in einem solchen Fall durch einen Strichpunkt getrennt sein müssen. Beachten Sie bitte weiter, daß ein Strichpunkt in einer Zeile mit einer Assembler-Anweisung, entgegen der Konventionen in Assembler, den Rest der Zeile nicht als Kommentar markiert. Kommentare müssen nach wie vor in der üblichen Pascal-Manier eingeleitet bzw. abgeschlossen werden, also mit { und } bzw. mit (* und *).

Einsatz der Register

In einer **asm**-Anweisung können Sie die Prozessor-Register in der gleichen Weise verwenden, wie Sie das von **external**-Prozeduren und -Funktionen her gewohnt sind. Sie haben lediglich dafür Sorge zu tragen, daß die Register BP, SP, SS und DS beim Austritt aus einem Assembler-Block, die gleichen Inhalte aufweisen wie zu Beginn der **asm**-Anweisung. Die Register AX, BX, CX, DX, SI, DI, ES und das Flag-Register stehen dagegen ohne Einschränkung zur Verfügung. Beim Eintritt in einen Assembler-Block können Sie von folgendem ausgehen: BP zeigt auf den aktuellen Stack-Rahmen, SP zeigt auf die Spitze des Stack, SS enthält die Segmentadresse des Stack-Segments und DS die Segmentadresse des Daten-Segments. Die anderen Register haben keinen definierten Inhalt.

Die Syntax einer Assembler-Anweisung

Assembler-Anweisungen haben folgende Syntax:

```
[ Label ":"]
äfix [ Opcode  [Operand "," Operand  ] ]
```

Label stellt einen Label-Bezeichner dar, *Präfix* ist ein Assemblerpräfixopcode (Operationscode), *Opcode* ist ein Assembler-Befehl oder eine Anweisung. *Operand* ist ein gültiger Assembler-Ausdruck.

Kommentare sind zwischen den einzelnen Assembler-Anweisungen erlaubt, jedoch nicht innerhalb einer Anweisung. Zulässig ist beispielsweise folgendes:

```
asm
  MOV   AX,1          {Startwert}
  MOV   CX,100        {Zähler}
end
```

Das nächste Programmfragment dagegen ist fehlerhaft:

```
asm
  MOV   {Startwert} AX,1
  MOV   CX, {Zähler} 100
end
```

Labels

Labels sehen in Assembler genauso aus wie in Pascal: ein Label-Bezeichner gefolgt von einem Doppelpunkt. Genau wie in Pascal muß ein Label, das in einer **asm**-Anweisung definiert ist, im **label**-Deklarationsteil des betreffenden Programmabschnittes deklariert sein. Von dieser Regel gibt es nur eine Ausnahme: *lokale Labels*.

Lokale Labels beginnen mit einem Klammeraffen (@) und sind, da Pascal-Bezeichner keinen Klammeraffen enthalten dürfen, automatisch auf die Verwendung innerhalb von **asm**-Anweisungen beschränkt. Der Gültigkeitsbereich erstreckt sich vom Schlüsselwort **asm** bis zum zugehörigen **end**.

Hinweis: Lokale Labels müssen, ganz wie in Turbo Assembler, nicht explizit definiert werden.

Der Bezeichner eines lokalen Labels besteht aus einem Klammeraffen (@) gefolgt von einem oder mehreren Buchtstaben (A..Z), Ziffern (0..9), Unterstrichen (_) oder weiteren Klammeraffen (@).

Der folgende Programmausschnitt zeigt die Verwendung normaler und lokaler Labels in **asm**-Anweisungen:

```
label Start, Stop
  ...
begin
  asm
```

```
    Start:
        ...
        JZ      Stop
    @1:
        ...
        LOOP    @1
  end;
  asm
    @1:
        ...
        JC      @2
        ...
        JMP     @1
    @2:
  end;
  goto Start;
  Stop:
end;
```

Wie dieser Programmausschnitt zeigt, kann ein normales Label innerhalb einer **asm**-Anweisung definiert werden. Verweise auf normale Labels können sowohl von außerhalb einer **asm**-Anweisung, wie auch von innerhalb erfolgen. Wie Sie weiter sehen, ist es zulässig, in verschiedenen **asm**-Anweisungen gleichlautende Bezeichner für lokale Labels zu benutzen, da ihr Gültigkeitsbereich ohnehin auf die **asm**-Anweisung beschränkt ist, in der sie verwendet werden.

Präfix-Opcodes

Der integrierte Assembler unterstützt folgende Präfixe:

LOCK	Bus lock
REP	wiederhole String-Operation
REPE/REPZ	wiederhole String-Operation solange Equal- bzw. Zero-Flag gesetzt ist
REPNE/REPNZ	wiederhole String-Operation solange Equal- bzw. Zero-Flag nicht gesetzt ist
SEGCS	CS-Register-Vorgabe
SEGDS	DS-Register-Vorgabe
SEGES	ES-Register-Vorgabe
SEGSS	SS-Register-Vorgabe

Ein Assembler-Befehl wird, soweit überhaupt, nur selten mit mehr als einem einzigen Präfix eingeleitet. Sinnvolle Kombinationen mit mehr als drei Präfixen (wie z.B. *LOCK SEGxx REPxx*) sind ohnehin kaum vorstellbar. Auf alle Fälle sollte man bei der Verwendung mehrerer Präfixe sehr vorsichtig sein (wichtig ist z.B. ihre Reihenfolge), da einige Prozessoren der 80x86-Familie manche Präfix-Kombinationen nicht korrekt verarbeiten. Tritt beispielsweise während einer

String-Operation mit vorangestelltem *REP*-Präfix ein Interrupt auf, so arbeitet die CPU die wiederholte String-Operation auch nach Beendigung des Interrupt korrekt ab. Tritt der Interrupt dagegen zu einem Zeitpunkt auf, wo zwar das *REP*-Präfix, der String-Befehl dagegen noch nicht eingelesen wurde, so »vergißt« ein 8088/8086-Prozessor das vorangegangene *REP*-Präfix.

Wie Sie sehen, ist es schon aus diesem Grund eine äußerst unsichere Sache, ein *LOCK*- und/oder ein *SEGxx*-Präfix zusammen mit einem *REP*-Präfix zu verwenden.

Hier einige Beispiele zur Verwendung von Präfixen:

```
asm
  REP    MOVSB        { kopiere CX Bytes von DS:SI nach ES:DI }
  SEGES  LODSW        { lade Word von ES:SI }
  SEGCS  MOV AX,[BX]  { identisch mit: MOV AX,CS:[BX] }
  SEGES               { bezieht sich auf die nächste Anweisung }
  MOV WORD PTR [DI],0 { wird zu: MOV WORD PTR ES:[DI],0 }
end;
```

Beachten Sie, daß ein Präfix in einer Anweisung auch ohne Assembler-Befehl vorkommen kann. In einem solchen Fall bezieht sich das Präfix auf den Assembler-Befehl der darauf folgenden Anweisung.

Befehlssätze

Der integrierte Assembler unterstützt alle 8086/8087- und 80286/80287-Befehle. Für den Einsatz der 8087-Befehle muß der Compiler-Schalter für den numerischen Koprozessor {$N+} eingeschaltet werden, die 80286-Befehle werden durch den Compiler-Schalter für den 80286-Prozessor {$G+} aktiviert. Sind beide Compiler-Schalter {$G+,N+} eingeschaltet, so sind damit auch die Befehle des 80287-Koprozessors verfügbar.

Eine komplette Beschreibung aller Assembler-Befehle finden Sie im Referenzhandbuch von Turbo Assembler 2.0 sowie in den Referenzbüchern der 80x86- und 80x87-Prozessoren.

Automatische Anpassung des RET-Befehls

Der *RET*-Befehl erzeugt, in Abhängigkeit vom Aufrufmodell der aktuellen Funktion oder Prozedur, für den Rücksprung automatisch einen Near- oder Far-Maschinencode.

```
procedure NearProc; near;
begin
  asm
    RET     { erzeugt einen Near-Rücksprung }
  end;
end;

procedure FarProc; far;
begin
  asm
```

```
    RET     { erzeugt einen Far-Rücksprung }
  end;
end;
```

Die Befehle *RETN* bzw. *RETF* erzeugen dagegen, unabhängig vom Aufrufmodell, gezielt einen Near- und Far-Rücksprung.

Automatische Sprunganpassung

Solange nicht anders spezifiziert, optimiert der integrierte Assembler Sprung-Befehle durch Einsetzen der kürzesten und damit effizientesten Befehlsvariante. Diese Optimierung erfolgt bei allen unbedingten Sprüngen (*JMP*), aber auch bei allen bedingten Sprüngen, wenn als Sprungziel ein Label, nicht jedoch eine Funktion oder Prozedur angegeben ist.

Liegt das Ziel-Label eines unbedingten Sprunges innerhalb einer Distanz von -128 bis +127 Bytes, dann generiert der integrierte Assembler einen kurzen Sprung (ein Byte Opcode und ein Byte Distanz); andernfalls wird ein Near-Sprung, bestehend aus einem Byte Opcode und zwei Bytes Distanz erzeugt.

Auch für einen bedingten Sprung wird ein kurzer Sprung generiert, wenn das Ziel-Label innerhalb einer Distanz von -128 bis +127 Bytes entfernt ist. Liegt das Sprungziel jedoch außerhalb dieses Bereiches, generiert der integrierte Assembler zwei Sprungbefehle: einen kurzen Sprung mit invertierter Sprungbedingung über einen darauffolgenden Near-Sprung, der zum eigentlichen Sprungziel führt. Diese Kombination besteht somit aus zwei Assembler-Befehlen und hat eine Länge von fünf Bytes. Für das Beispiel

```
JC      Stop
```

erzeugt der integrierte Assembler folgende Sequenz (wobei *Stop* außerhalb der für kurze Sprünge erreichbaren Distanz liegt):

```
JNC     Skip
JMP     Stop
Skip:
```

Sprünge auf Funktionen und Prozeduren werden immer entweder als Near oder Far erzeugt, niemals aber als kurze (short) Sprünge. Außerdem erlaubt der integrierte Assembler keine bedingten Sprünge auf Funktionen oder Prozeduren. Möglich ist es hingegen, über die Konstrukte *NEAR PTR* bzw. *FAR PTR*, gezielt einen Near- oder Far-Sprung auf ein Label zu erzwingen, selbst wenn das Sprungziel durch einen kurzen Sprung erreichbar wäre. Zum Beispiel wird, ungeachtet der tatsächlichen Sprungdistanz, für

```
JMP     NEAR PTR Stop
JMP     FAR PTR Stop
```

immer ein Near-Sprung bzw. ein Far-Sprung erzeugt.

Assembler-Anweisungen

Der integrierte Assembler von Turbo Pascal unterstützt drei verschiedene Anweisungen zur Erzeugung von Datenstrukturen: *DB* (define byte), *DW* (define word) und *DD* (define double word). Diese drei Anweisungen erzeugen Daten, entsprechend der nachfolgenden, durch Kommata voneinander getrennten Operanden.

Die *DB*-Anweisung generiert eine Folge von Daten-Bytes. Jeder Operand muß ein konstanter Ausdruck zwischen -128 und 255 oder eine Zeichenkette beliebiger Länge sein. Konstanten-Ausdrücke erzeugen Code von einem Byte Länge, Zeichenketten ergeben eine Byte-Folge, wobei der Wert eines jeden Bytes dem ASCII-Code des angegebenen Zeichens entspricht.

Die *DW*-Anweisung generiert eine Folge von Daten-Words. Jeder Operand muß ein konstanter Ausdruck zwischen -32768 und 65535 oder ein Adreß-Ausdruck sein. Im Falle eines Adreß-Ausdruckes erzeugt der integrierte Assembler ein Word, dessen Inhalt einem Near-Zeiger, also dem Offset-Anteil der angegebenen Adresse, entspricht.

Die *DD*-Anweisung generiert eine Folge von Double-Words. Jeder Operand muß ein konstanter Ausdruck zwischen -2147483648 und 4294967295 oder ein Adreß-Ausdruck sein. Im Falle eines Adreß-Ausdruckes erzeugt der integrierte Assembler einen Far-Zeiger; d.h. es wird ein vollständiger 32-Bit-Zeiger mit Offset- und Segment-Anteil abgelegt.

Solchermaßen erzeugte Daten werden, genauso wie auch der vom integrierten Assembler generierte Programmcode, immer im Code-Segment abgelegt. Im Daten-Segment werden auch weiterhin nur diejenigen initialisierten oder uninitialisierten Daten abgelegt, die über eine entsprechende Pascal-Deklaration (**const** oder **var**) deklariert wurden.

Hier einige Beispiele für die verschiedenen Anweisungen:

```
asm
  DB     0FFH                      { ein Byte }
  DB     0,99                      { zwie Bytes }
  DB     'A'                       { Ord(A) }
  DB     'Hallo Welt...',0DH,0AH   { String und CR/LF }
  DB     12,"Turbo Pascal"         { String wie in Pascal }
  DW     0FFFFH                    { ein Word }
  DW     0,9999                    { zwei Words }
  DW     'A'                       { entspricht DB   'A',0 }
  DW     'BA'                      { entspricht DB   'A','B' }
  DW     MeineVar                  { Offset von MeineVar }
  DW     MeineProc                 { Offset von MeineProc }
  DD     0FFFFFFFFH                { ein Double-Word }
  DD     0,999999999               { zwei Double-Words }
  DD     'A'                       { entspricht
                                     DB   'A',0,0,0 }
  DD     'DCBA'                    { entspricht
                                     DB   'A','B','C','D' }
```

```
  DD     MeineVar                  { Zeiger auf MeineVar }
  DD     MeineProc                 { Zeiger auf MeineProc }
end;
```

In Turbo Assembler wird durch das Vorstellen eines Bezeichners vor eine *DB*-, *DW*- oder *DD*-Anweisung eine Variable entsprechender Größe an der Speicherstelle der Anweisung deklariert. Turbo Assembler erlaubt z.B. folgendes:

```
ByteVar   DB  ?
WordVar   DW  ?
...
  MOV    AL,ByteVar
  MOV    BX,WordVar
```

Der integrierte Assembler unterstützt solche Deklarationen nicht, die einzigen Symbole, die man innerhalb einer **asm**-Anweisung definieren kann, sind Labels. Variablen müssen auch künftig gemäß der Pascal-Syntax im Pascal-Deklarationsteil deklariert werden. Das eben gezeigte Beispiel wird im integrierten Assembler folgendermaßen formuliert:

```
var
  ByteVar:    Byte;
  WordVar:    Word;
...
asm
  MOV    AL,ByteVar
  MOV    BX,WordVar
end;
```

Operanden

Operanden sind Ausdrücke, bestehend aus einer Kombination von Konstanten, Registern, Symbolen und Operatoren. Obwohl die Syntax des integrierten Assemblers in vielen Fällen der Pascal-Syntax angelehnt wurde, gibt es doch eine Reihe von Unterschieden, die im folgenden erklärt werden.

Nachstehende reservierte Wörter haben, soweit sie als Operanden vorkommen, für den integrierten Assembler eine vordefinierte Bedeutung:

AL	CH	DX	OR	ST
AND	CS	ES	PTR	TBYTE
AX	CX	HIGH	QWORD	TYPE
BH	DH	LOW	SHL	WORD
BL	DI	MOD	SHR	XOR
BP	DL	NEAR	SI	
BX	DS	NOT	SP	
BYTE	DWORD	OFFSET	SS	

Diese reservierten Wörter haben immer Vorrang vor selbstdefinierten Bezeichnern. So wird zum Beispiel im folgenden Code-Auschnitt

```
var
  CH:    Char;
...
asm
  MOV    CH,1
end;
```

der Wert 1 *nicht* der Variablen CH, sondern dem Register *CH* zugewiesen. Um die Variable CH anzusprechen, muß dem Variablennamen das Et-Zeichen (&) vorangestellt werden:

```
asm
  MOV    &CH,1
end;
```

Vermeiden Sie möglichst Bezeichnernamen, die mit reservierten Wörtern identisch sind, denn eine Verwechslung von Bezeichnern führt sehr leicht zu kaum auffindbaren Fehlern.

Ausdrücke

Ausdrücke werden im integrierten Assembler immer als 32-Bit-Werte berechnet. Fließkomma-Zahlen und Strings, ausgenommen String-Konstanten, werden nicht akzeptiert.

Die Ausdrücke des integrierten Assemblers setzen sich aus *Ausdruck-Elementen* und *Operatoren* zusammen. Außerdem hat jeder Ausdruck eine *Ausdruck-Klasse* und einen *Ausdruck-Typ*. Dieses Konzept wird in den folgenden Abschnitten noch näher erläutert.

Unterschiede zwischen Pascal- und Assembler-Ausdrücken

Der wichtigste Unterschied zwischen Ausdrücken in Turbo Pascal und Ausdrücken im integrierten Assembler besteht darin, daß Ausdrücke im integrierten Assembler stets einen *konstanten Wert* haben müssen. Mit anderen Worten: der Wert eines Ausdrucks muß zum Zeitpunkt der Compilierung feststehen. Das folgende Beispiel soll dies verdeutlichen. Wenn

```
const
  X = 10;
  Y = 20;
var
  Z : Integer;
```

eine gegebene Deklaration ist, ergibt folgende Zeile eine gültige Assembler-Anweisung:

```
asm
  MOV     Z,X+Y
end;
```

Da sowohl *X* als auch *Y* Konstanten sind, ist es faktisch gleichgültig, ob im letzten Beispiel der Ausdruck *X+Y*, oder stattdessen die Konstante 30 angegeben ist. Der Ausdruck *X+Y* stellt lediglich eine zweckdienlichere Schreibweise ein und derselben Konstanten dar. Der aus dieser Anweisung resultierende Maschinen-Befehl ist ein Ladebefehl, durch den die Variable *Z* mit dem Wert 30 geladen wird. Verwendet man anstelle der Konstanten *X* und *Y* allerdings Variablen, wird der Unterschied zwischen Pascal und Assembler deutlich:

```
var
  X, Y : Integer;
```

Die Summe aus *X* und *Y* ist zum Zeitpunkt der Compilierung nicht bekannt. Der Assembler-Block zur Berechnung der Summe aus *X+Y* muß daher wie folgt geändert werden:

```
asm
  MOV   AX,X
  ADD   AX,Y
  MOV   Z,AX
end;
```

Ein weiterer wichtiger Unterschied zwischen Pascal und Ausdrücken im integrierten Assembler betrifft die Interpretation von Variablen. Während sich eine Variablenreferenz in Pascal immer auf den *Inhalt* der Variablen bezieht, betrachtet der integrierte Assembler eine Variablenreferenz als eine Referenz auf die *Adresse* der Variablen. Ein Ausdruck wie *X+4* stellt in Pascal die Summe des Inhalts der Variablen *X* und der Konstanten 4 dar. In Assembler bezieht sich der Ausdruck *X+4* dagegen auf den Inhalt des Word an der Speicheradresse, die vier Bytes höher liegt als die Adresse der Variablen *X*. Mit folgender, erlaubter Assembler-Anweisung

```
asm
  MOV   AX,X+4
end;
```

wird das AX-Register also *nicht* mit der Summe *X+4* geladen, sondern mit dem Inhalt des Word, das vier Bytes oberhalb von *X* gespeichert ist. Die korrekte Anweisung für die Addition sieht so aus:

```
asm
  MOV   AX,X
  ADD   AX,4
end;
```

Ausdruck-Elemente

Die grundlegenden Elemente eines Ausdrucks sind *Konstanten*, *Register* und *Symbole*.

Konstanten

Der integrierte Assembler unterstützt zwei verschiedene Arten von Konstanten: *numerische Konstanten* und *String-Konstanten*.

Numerische Konstanten

Numerische Konstanten sind ganze Zahlen im Wertebereich von 2147483649 bis 4294967295.

Gemäß Standardeinstellung werden numerische Konstanten in dezimaler Schreibweise geschrieben (Basis 10). Der integrierte Assembler unterstützt allerdings auch die binäre (Basis 2), die oktale (Basis 8) und die hexadezimale (Basis 16) Schreibweise. Die Wahl der Zahlenbasis erfolgt durch das Anhängen eines Suffix an die betreffende Zahl: Suffix *B* für binär, Suffix *O* für oktal und Suffix *H* (bzw. Voranstellen des Präfix *$*, wie in Pascal-Ausdrücken) für hexadezimal.

Vorsicht! Für Pascal-Ausdrücke bleibt alles beim Alten: Zur Verfügung stehen ausschließlich das dezimale und das hexadezimale Zahlensystem; Hexadezimal-Zahlen werden immer mit dem Präfix *$* eingeleitet.

Numerische Konstanten müssen mit einer Ziffer von 0 bis 9 beginnen bzw. mit dem Präfix $ bei hexadezimalen Zahlen. Verwendet man bei hexadezimalen Zahlen das Suffix H, muß auch diese Zahl mit einer Ziffer von 0 bis 9 beginnen (z.B. 0F5Fh). Diese Schreibweise ist notwendig, um hexadezimele Zahlen von Bezeichnern unterscheiden zu können.

String-Konstanten

String-Konstanten müssen entweder von Apostrophen ('...') oder Anführungszeichen ("...") eingeschlossen werden. Soll ein den String einschließendes Zeichen selbst Bestandteil des Strings sein, so muß es innerhalb des Strings doppelt angegeben werden. Hier einige Beispiele von String-Konstanten:

```
'Z'
'Turbo Pascal'
"Das war's"
'"Das war''s",sagte er.'
'100'
'"'
"'"
```

Beachten Sie, daß in der vierten Zeichenkette zwei aufeinanderfolgende Apostrophen verwendet werden, um einen einzigen Apostrophen als String-Bestandteil zu erzeugen.

String-Konstanten, die über *DB*-Anweisungen deklariert sind, haben keine Längenbeschränkung und ergeben eine Byte-Folge, welche den ASCII-Werten der einzelnen Zeichen entspricht. Werden String-Konstanten allerdings mit den

Anweisungen *DW* oder *DD* erzeugt, dann dürfen diese Konstanten eine Länge von zwei bzw. vier Bytes nicht überschreiten. Die beiden letzten Anweisungen bezeichnen einen numerischen Wert, der auch Teil eines Ausdruckes sein kann und sich sich wie folgt berechnen läßt:

```
Ord(Ch1) + Ord(Ch2) shl 8 + Ord(Ch3) shl 16 + Ord(Ch4) shl 24
```

Z1 entspricht dem letzten Zeichen, *Z4* dem ersten Zeichen der Zeichenkette. Ist die Zeichenkette kürzer als vier Zeichen, werden die entsprechenden fehlenden Bytes auf null gesetzt. Hier einige Beispiele von String-Konstanten und deren numerischer Repräsentation:

```
'a'     00000061H
'ba'    00006261H
'cba'   00636261H
'dcba'  64636261H
'a '    00006120H
'a'*2   000000E2H
'a'-'A' 00000020H
not 'a' FFFFFF9EH
```

Register
Die folgenden reservierten Symbole bezeichnen die CPU-Register:

allgemeine 16-Bit-Register	AX	BX	CX	DX
niederwertige 8-Bit-Register	AL	BL	CL	DL
höherwertige 8-Bit-Register	AH	BH	CH	DH
16-Bit-Zeiger-Register	SP	BP	SI	DI
16-Bit-Segment-Register	CS	DS	SS	ES
80x87-Register-Stack	ST			

Wenn ein Operand aus nur einem Register-Bezeichner besteht, nennt man diesen Operanden einen Register-Operanden. Alle Register können als Register-Operanden dienen. Darüber hinaus können manche Register noch in einem anderen Zusammenhang verwendet werden.

- Die Basis-Register (BP und BX), ebenso wie die Index-Register (SI und DI) können – in eckigen Klammern geschrieben – für indizierte Speicherzugriffe verwendet werden. Folgende Kombinationen von Basis- und Index-Registern sind erlaubt: [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], [BP+DI].

- Die Segment-Register (ES, CS, SS und DS) - gefolgt von einem Doppelpunkt – können zur Überschreibung der Segmentvorgabe benutzt werden.

- Das Symbol *ST* bezeichnet die Spitze des Koprozessor-Register-Stack. Jedes der acht Koprozessor-Register kann durch das Symbol *ST* gefolgt von der Registernummer in runden Klammern (*ST(x)*) angesprochen werden. Die Register sind von 0 bis 7 numeriert, wobei die Nummer die Distanz zur Spitze des Register-Stack angibt.

Symbole

Der integrierte Assembler ermöglicht den Zugriff auf fast alle Pascal-Symbole. Diese Symbole können Labels, Konstanten, Typ-Bezeichner, Variablen, Prozeduren und Funktionen sein. Zusätzlich stellt der integrierte Assembler noch folgende spezielle Symbole zu Verfügung:

@Code *@Data* *@Result*

Die Symbole *@Code* und *@Data* repräsentieren das aktuelle Code- bzw. Daten-Segment. Sie sollten nur zusammen mit dem *SEG*-Operator verwendet werden:

```
asm
  MOV   AX,SEG @Data
  MOV   DS,AX
end;
```

Über das Symbol *@Result* können Sie innerhalb des Anweisungsteils einer Funktion auf das Ergebnis dieser Funktion zugreifen. In der Funktion

```
function Sum (X, Y: Integer): Integer;
begin
  Sum := X + Y;
end;
```

wird der Wert des Funktionsergebnisses der Variablen *Sum* zugewiesen. Wäre diese Funktion in Assembler geschrieben, müßten Sie das Funktionsergebnis über *@Result* zurückliefern:

```
function Sum (X, Y: Integer): Integer;
begin
  asm
    MOV   AX,X
    ADD   AX,Y
    MOV   @Result,AX
  end;
end;
```

In diesem Beispiel mag die Verwendung des *@Result*-Symbols überflüssig erscheinen, da der Rückgabewert einer Integer-Funktion sowieso im *AX*-Register erwartet wird. Bei normalen Funktionen wird jedoch im Eintrittscode der Funktion Platz für das Funktionsergebnis auf dem Stack reserviert. Beim Austritt wird der Inhalt dann in die entsprechenden Register kopiert und steht dem Aufrufer zur Verfügung. Mit anderen Worten ist die Verwendung des *@Result*-Symbols absolut notwendig (vergl. Assembler-Funktionen).

Folgende Symbole können nicht in Assembler-Ausdrücken verwendet werden:

- Standard-Prozeduren und -Funktionen (z.B. *WriteLn*, *Chr*)
- Die vordefinierten Pseudo-Arrays *Mem*, *MemW*, *MemL*, *Port* und *PortW*
- String-, Fließkomma- und Mengen-Konstanten

- Prozeduren und Funktionen, die mit der **Inline**-Anweisung deklariert sind
- Labels, die nicht im aktuellen Block definiert sind
- Das Symbol *@Result* außerhalb von Funktionen

Tabelle 22.1 gibt eine Zusammenfassung der Werte, Klassen und Typen der verschiedenen Arten von Symbolen wieder, die in Assembler-Ausdrücken zulässig sind (Ausdruck-Klassen und -Typen werden in einem der folgenden Abschnitten behandelt).

Tabelle 22.1 Symbol-Werte, -Klassen und -Typen

Symbol	Wert	Klasse	Typ
Label	Label-Adresse	Speicher	Short
Konstante	Konstantenwert	Konstante	0
Type	0	Speicher	wie SizeOf(Type)
Feld	Feld-Offset	Speicher	wie SizeOf(Feld)
Variable	Variablen-Adresse	Speicher	wie SizeOf(Variable)
Prozedur	Prozedur-Adresse	Speicher	NEAR oder FAR
Funktion	Funktion-Adresse	Speicher	NEAR oder FAR
Unit	0	Konstante	0
@Code	Adresse des Code-Segments	Speicher	0FFF0H
@Data	Adresse des Daten-Segments	Speicher	0FFF0H
@Result	Offset der Result-Variablen	Speicher	

Lokale Variablen (Variablen, die in Funktionen und Prozeduren deklariert sind) werden immer auf dem Stack angelegt und relativ zu SS:BP adressiert. Der Wert des Symbols einer lokalen Variablen gibt den vorzeichenbehafteten Offset relativ zu SS:BP an. Bei Referenzen auf lokale Variablen fügt der Assembler automatisch den Bezug auf das Register BP hinzu. So übersetzt der integrierte Assembler bei folgender Deklaration

```
procedure Test;
var
  Count: Integer;
```

die **asm**-Anweisung

```
asm
  MOV   AX,Count
end;
```

in folgenden Assembler-Befehl:

```
MOV AX,[BP-2]
```

Der integrierte Assembler behandelt **var**-Parameter immer als 32-Bit-Zeiger mit einer Größe von vier Bytes. In Pascal gibt es keinen Unterschied zwischen dem Zugriff auf **var**-Parameter und dem auf Werteparameter. Im integrierten Assembler ist das anders. Da es sich bei **var**-Parametern tatsächlich um Zeiger

handelt, müssen sie im integrierten Assembler auch als solche behandelt werden. Wenn Sie also auf den Inhalt eines **var**-Parameters zugreifen möchten, müssen Sie zunächst den 32-Bit-Zeiger laden und dann auf die Speicherstelle zugreifen, auf die der Zeiger verweist. Wären die Parameter *X* und *Y* aus der obigen Funktion *Sum* nun **var**-Parameter, ergäbe dies folgenden Programm-Code:

```
function Sum (var X, Y: Integer): Integer;
begin
  asm
    LES   BX,X
    MOV   AX,ES:[BX]
    LES   BX,Y
    ADD   AX,ES:[BX]
    MOV   @Result,AX
  end;
end;
```

Manche Symbole, wie beispielsweise Record-Typen und -Variablen, markieren einen Speicherbereich, dessen Felder mit einem Punkt und dem Feldbezeichner angesprochen werden können. Bei der folgenden Deklaration

```
type
  Point = record
    X, Y: Integer;
  end;
  Rect = record
    A, B: Point;
  end;

var
  P: Point;
  R: Rect;
```

können die Felder der Variablen P und R wie folgt angesprochen werden:

```
asm
  MOV  AX,P.X
  MOV  DX,P.Y
  MOV  CX,R.A.X
  MOV  BX,R.B.Y
end;
```

Typ-Bezeichner können bei Speicheroperationen zur Erzeugung des richtigen Offsets eingesetzt werden. Für die folgenden fünf Anweisungen wird ein und derselbe Maschinen-Code erzeugt, der den Inhalt der Speicherstelle ES:[DI+4] in das AX-Register lädt.

```
asm
  MOV  AX,(Rect PTR ES:[DI]).B.X
  MOV  AX,Rect(ES:[DI]).B.X
```

```
  MOV  AX,ES:Rect[DI].B.X
  MOV  AX,Rect[ES:DI].B.X
  MOV  AX,ES:[DI].Rect.B.X
end;
```

Der Speicherbereich eines Record- bzw. Objekt-Typs wird von dessen Typ-, Feld- und Variablen-Symbolen beschrieben. Zusätzlich kann man mit einem Unit-Bezeichner, in gleicher Weise wie in Pascal, auf ein Symbol dieser Unit zugreifen.

Ausdruck-Klassen

Der Integrierte Assembler kennt drei unterschiedliche Klassen von Ausdrücken: *Register-Ausdrücke, Speicher-Referenzen* und *konstante Ausdrücke*.

Ein Ausdruck, der nur aus einem Registernamen besteht, ist ein Register-Ausdruck. Beispiele für Register-Ausdrücke sind *AX*, *CL*, *DI* und *ES*. Register-Ausdrücke, die als Operanden verwendet werden, weisen den Assembler an, Befehle zu generieren, die mit den CPU-Registern operieren.

Ausdrücke, die auf Speicherstellen verweisen, sind Speicher-Referenzen; zu dieser Kategorie gehören Pascal-Labels, Variablen, typisierte Konstanten, Prozeduren und Funktionen.

Ausdrücke, die weder Register- noch Speicher-Ausdrücken angehören, sind konstante Ausdrücke; in diese Gruppe gehören untypisierte Konstanten und Typ-Bezeichner.

Werden konstante Ausdrücke und Speicher-Ausdrücke als Operanden verwendet, dann wird für sie unterschiedlicher Code erzeugt.

```
const
  Start = 10;

var
  Count: Integer;

...
asm
  MOV  AX,Start             {MOV  AX,xxxx}
  MOV  BX,Count             {MOV  BX,[xxxx]}
  MOV  CX,[Start]           {MOV  CX,[xxxx]}
  MOV  DX,OFFSET Count      {MOV  DX,xxxx}
end;
```

Da es sich bei *Start* um einen konstanten Ausdruck handelt, erzeugt der Assembler einen Befehl, der das Register mit einem konstanten Wert lädt (move immediate). Die zweite *MOV*-Anweisung dagegen wird in eine Anweisung für einen Speicherzugriff (move memory) übersetzt, da *Count* eine Speicher-Referenz darstellt. Bei der dritten *MOV*-Anweisung bewirken die eckigen Klammern, daß *Start* als eine Speicher-Referenz behandelt wird (Inhalt der Speicherstelle mit Offset 10). In der vierten *MOV*-Anweisung schließlich wird *Count* mittels

des vorangestellten *OFFSET*-Operators in einen konstanten Ausdruck (Offset von *Count* im Daten-Segment) verwandelt.

Wie Sie sehen, haben der *OFFSET*-Operator und die eckigen Klammern eine zueinander komplementäre Wirkung. Der aus den beiden folgenden Assembler-Befehlen resultierende Maschinen-Code, ist daher identisch mit dem Maschinen-Code der beiden ersten Assembler-Befehle aus dem letzten Beispiel:

```
asm
  MOV  AX,OFFSET [Start]
  MOV  BX,[OFFSET Count]
end;
```

Speicher-Ausdrücke und konstante Ausdrücke unterteilen sich zusätzlich in *relozierbare Ausdrücke* und *absolute Ausdrücke*. Ein relozierbarer Ausdruck erfordert noch eine *Relokation* während des Link-Vorgangs, ein absoluter Ausdruck hingegen nicht. Ausdrücke, die sich auf Prozeduren, Funktionen, Variablen und Labels beziehen, sind relozierbare Ausdrücke. Beinhaltet ein Ausdruck dagegen ausschließlich einen konstanten Wert, so handelt es sich um einen absoluten Ausdruck.

Relokation: Zur Übersetzungszeit ist dem Compiler die endgültige Adresse einer Prozedur, Funktion, Variablen oder eines Labels nicht bekannt. Dieser Zustand ändert sich erst, wenn der Linker Symbolen eine spezifische absolute Adresse zuweist.

Der integrierte Assembler erlaubt beliebige Operationen mit absoluten Ausdrücken, begrenzt aber die Operationen mit relozierbaren Ausdrücken auf die Addition bzw. Subtraktion von Konstanten.

Ausdruck-Typen

Jeder Ausdruck des integrierten Assemblers hat einen bestimmten Typ oder, genauer gesagt, eine bestimmte Größe, da der Assembler den Typ eines Ausdrucks als eine Größenangabe für den Speicherzugriff interpretiert. Zum Beispiel belegt eine *Integer*-Variable zwei Bytes im Speicher und demzufolge hat der Betrag ihrer Größe (ihres Typs) den Wert zwei.

Der integerierte Assembler führt Typ-Prüfungen durch, wo immer dies möglich ist. In den folgenden Anweisungen

```
var
  QuitFlag: Boolean;
  OutBufPtr: Word;
...
asm
  MOV  AL,QuitFlag
  MOV  BX,OutBufPtr
end;
```

prüft der Assembler, ob die Größe von *QuitFlag* ein Byte und die Größe von *OutBufPtr* zwei Bytes beträgt. Stellt der Assembler unterschiedliche Typen

(Größen) fest, resultiert daraus eine Fehlermeldung. Folgendes ist zum Beispiel nicht erlaubt

```
asm
  MOV  DL,OutBufPtr
end;
```

da es sich bei DL um ein 8-Bit-Register (1 Byte) handelt, *OutBufPtr* jedoch eine Größe von zwei Bytes hat. Allerdings kann der Typ eines Speicher-Ausdrucks durch entsprechende Operatoren geändert werden. Korrekte Schreibweisen für das letzte Beispiel sind

```
asm
  MOV  DL,BYTE PTR OutBufPtr
  MOV  DL,BYTE (OutBufPtr)
  MOV  DL,OutBufPtr.Byte
end;
```

wobei sich alle Zugriffe auf das erste (niederwertige) Byte von *OutBufPtr* beziehen.

In manchen Fällen ist ein Speicher-Ausdruck untypisiert. Ein Beispiel dafür ist ein konstanter Ausdruck in eckigen Klammern:

```
asm
  MOV  AL,[100H]
  MOV  BX,[100H]
end;
```

Der integrierte Assembler erlaubt beide Anweisungen, da der Ausdruck [100H] nicht typisiert ist. Der Ausdruck [100H] stellt lediglich einen Bezug auf den Inhalt der Adresse 100H im Daten-Segment dar, der Ausdruck-Typ ist durch das verwendete Register (AL für das Byte und BX für das Word) bestimmt. Situationen, in denen der Assembler keine Typ-Feststellung über einen zweiten Operator durchführen kann, erfordern eine explizite Typangabe:

```
asm
  INC   BYTE PTR [100H]
  IMUL  WORD PTR [100H]
end;
```

Die Tabelle 22.2 faßt die vordefinierten Typ-Symbole zusammen, die der integrierte Assembler zusätzlich zu den in Pascal deklarierten zur Verfügung stellt.

Tabelle 22.2 Vordefinierte Typ-Symbole

Symbol	Typ	Symbol	Typ
BYTE	1	TBYTE	10
WORD	2	NEAR	0FFFFH
DWORD	4	FAR	0FFFEH
QWORD	8		

Beachten Sie insbesondere die Pseudo-Typen **near** und **far**, die bei Prozedur- und Funktions-Symbolen zur Angabe des Aufrufmodells dienen. Ebenso wie andere Symbole auch, können Sie in Typangaben **near** und **far** verwenden.

FarProc sei zum Beispiel eine **far**-Prozedur:

```
procedure FarProc; far;
```

Wenn Sie nun im selben Modul Assembler-Code schreiben, ist es möglich, die weit effizientere **near**-Anweisung für den Prozedur-Aufruf einzusetzen:

```
asm
  PUSH  CS
  CALL  NEAR PTR FarProc
end;
```

Ausdruck-Operatoren

Der integrierte Assembler verfügt über eine Vielzahl von Operatoren. Die Rangfolge dieser Operatoren unterscheidet sich jedoch erheblich von der Rangfolge in Pascal. So hat zum Beispiel der **and**-Operator einen niedrigeren Rang als die Plus(+)- und Minus(-)-Operatoren; für Pascal gilt genau das Gegenteil: hier hat der **and**-Operator Vorang. Im folgenden werden die Operatoren des integrierten Assemblers gemäß ihrer Priorität in absteigender Reihenfolge aufgelistet.

& »Bezeichner-Vorgabe« – der Bezeichner nach dem Et-Zeichen wird als benutzerdefiniertes Symbol behandelt, selbst wenn ein vordefiniertes Symbol gleichen Namens im integrierten Assembler exisitiert.

(...) »Teilausdrücke« – Ausdrücke innerhalb runder Klammern werden komplett ausgewertet und als einziges Ausdruck-Element behandelt. Ein weiterer Ausdruck kann diesem Teilausdruck vorangestellt werden. Das Ergebnis ergibt sich aus der Summe beider Ausdrücke, wobei der Ergebnis-Typ dem Typ des vorangestellten Ausdruckes entspricht.

[...] »Speicher-Referenzen« – der Ausdruck innerhalb der eckigen Klammern wird komplett ausgewertet und als einzelnes Ausdruck-Element behandelt. Der Klammerausdruck kann für indizierte Speicherzugriffe eine Kombination der Register *BX*, *BP*, *SI* oder *DI* und des Plus (+)- bzw. Minus (-)-Operators enthalten. Ist dem Klammerausdruck ein weiterer Ausdruck vorangestellt, ergibt sich das Ergebnis aus der Summe beider Ausdrücke. Das Ergebnis ist immer eine Speicher-Referenz und hat den gleichen Typ wie der vorangestellte Ausdruck.

. »Strukturauswahl« – dieser Operator findet, genau wie in Pascal, beim Zugriff auf Felder eines Records Anwendung. Der Typ des Gesamtausdruckes entspricht dem Typ des Ausdruckes nach dem Punkt, also dem Typ des Feldes auf das der Zugriff erfolgt.

HIGH Liefert die acht höherwertigen Bits des konstanten Ausdruckes, der dem

Operator folgt, zurück. Die Größe des Ausdruckes muß dem Datentyp Word entsprechen.

`LOW` Liefert die acht niederwertigen Bits des konstanten Ausdruckes, der dem Operator folgt, zurück. Die Größe des Ausdruckes muß dem Datentyp Word entsprechen.

`+` »Unäres Plus« – liefert den folgenden Ausdruck unverändert zurück. Der Operator dient lediglich zur expliziten Darstellung positiver Zahlen.

`-` »Unäres Minus« – liefert den negierten Wert eines Ausdruckes zurück. Der Ausdruck muß ein absoluter, konstanter Wert sein.

`:` »Segment-Vorgabe« – weist den integrierten Assembler an, die Adresse des folgenden Ausdruckes relativ zu einem bestimmten Segment zu berechnen. Das Segment ist durch den Register-Namen (CS, DS, SS oder ES) vor dem Doppelpunkt bestimmt. Das Ergebnis ist eine Speicher-Referenz und wird aus dem Ausdruck nach dem Doppelpunkt errechnet. Bei einer Segment-Vorgabe erzeugt der integrierte Assembler einen Präfix, um die CPU anzuweisen, den folgenden Ausdruck auf ein anderes als das voreingestellte Segment-Register zu beziehen.

`OFFSET` liefert den Offset-Anteil des folgenden Ausdruckes zurück. Das Ergebnis ist eine Konstante.

`SEG` liefert den Segment-Anteil des folgenden Ausdruckes zurück. Das Ergebnis ist eine Konstante.

`TYPE` Liefert den Typ (Größe in Bytes) des folgenden Ausdruckes zurück. Bei einer Konstanten hat das Ergebnis den Wert 0. Dieser Operator kann mit der *SizeOf*-Funktion von Pascal verglichen werden.

`PTR` »Typecast-Operator« – weist den integrierten Assembler an, den Datentyp des folgenden Ausdruckes durch einen anderen zu ersetzen. Der neue Datentyp entspricht dem des Ausdruckes vor dem Operator. Das Ergebnis ist eine Speicher-Referenz, deren Wert bestimmt ist durch den Ausdruck nach dem Operator.

`*` »Multiplikation« – beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

`/` »Ganzzahlige Division« – beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

`MOD` »Modulo-Operator« – Rest einer ganzzahligen Division. Beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

`SHL` »Logisches Linksschieben« – beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

`SHR` »Logisches Rechtsschieben« – beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

`+` »Addition« – die Ausdrücke können konstante Werte oder Speicher-Referenzen sein. Allerdings darf nur einer der beiden Ausdrücke ein relozierbarer

Wert sein. Das Ergebnis ist in jedem Fall ein relozierbarer Wert, wenn ein Ausdruck relozierbar ist. Sind beide Ausdrücke Speicher-Referenzen, dann ist auch das Ergebnis eine Speicher-Referenz.

- »Subtraktion« – die Klasse des ersten Ausdruckes kann beliebig sein, der zweite Ausdruck muß ein absoluter, konstanter Wert sein. Die Klasse des Ergebnisses entspricht der Klasse des ersten Ausdruckes.

NOT »Bitweise Negation« – der Ausdruck muß ein absoluter, konstanter Ausdruck sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Ausdruck.

AND »Bitweise Und-Verknüpfung« – beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

OR »bitweise Oder-Verknüpfung« – beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

XOR Bit-weise Exklusiv-Oder-Verknüpfung« – beide Ausdrücke müssen absolute, konstante Werte sein. Das Ergebnis ist ebenfalls ein absoluter, konstanter Wert.

Assembler-Prozeduren und Funktionen

Neben der Möglichkeit, **asm**-Anweisungen (**asm** ... **end**) innerhalb normaler **begin**...**end**-Anweisungen zu verwenden, unterstützt der integrierte Assembler auch sog. **assembler**-Prozeduren und -Funktionen. Diese Routinen benötigen keinen **begin**...**end**-Block, sondern nur einen **asm**...**end**-Block. Hier ein Beispiel einer **assembler**-Funktion:

```
function LongMul(X,Y: Integer): LongInt; assembler;
asm
  MOV   AX,X
  IMUL  Y
end;
```

Die **assembler**-Anweisung veranlaßt Turbo Pascal, eine Reihe von Optimierungen vorzunehmen:

- Der Compiler erzeugt keinen Code, der Werte-Parameter in lokale Variablen kopiert. Dies betrifft sämtliche String-Werte-Parameter, sowie alle Werte-Parameter, deren Größe nicht 1, 2 oder 4 Bytes beträgt. Solche Parameter müssen innerhalb einer Prozedur bzw. Funktion behandelt werden, als seien sie **var**-Parameter.

- Der Compiler reserviert keinen Speicherplatz für die Funktionsergebnis-Variable – eine Referenz auf das *@Result*-Symbol verursacht einen Fehler. Die einzige Ausnahme von dieser Regel betrifft String-Funktionen, denn bei solchen Funktionen wird das *@Result*-Symbol immer dazu verwendet, einen Zeiger aufzunehmen, der durch die aufrufende Routine alloziert wird.

- Der Compiler erzeugt für Prozeduren und Funktionen, die weder Parameter noch lokale Variablen haben, keinen Stack-Rahmen.

- Der automatisch generierte Eintritts- und Austritts-Code einer **assembler**-Prozedur bzw. -Funktion sieht folgendermaßen aus (die einzelnen Anweisungen werden nur dann erzeugt, wenn die nebenstehende Bedindung erfüllt ist):

```
PUSH   BP              {wenn Locals  0 oder Params  0}
MOV    BP,SP           {wenn Locals  0 oder Params  0}
SUB    SP,Locals       {wenn Locals  0}
...
MOV    SP,BP           {wenn Locals  0}
POP    BP              {wenn Locals  0 oder Params  0}
RET    Params          {immer}
```

Locals gibt die Größe der lokalen Variablen an und *Params* die Größe der Parameter. Haben sowohl *Locals* wie auch *Params* den Wert null, erzeugt der integrierte Assembler keinen Eintritts-Code, und der Austritts-Code besteht lediglich aus dem *RET*-Befehl.

Bei **assembler**-Funktionen erfolgt die Rückgabe des Funktionsergebnisses auf die folgende Weise:

- Ordinal-Typen (Integer-, Char- und Aufzählungs-Typen) werden in AL (bei 8-Bit Werten), in AX (bei 16-Bit Werten) und in DX:AX (bei 32-Bit Werten) zurückgegeben.

- Real-Typen werden in DX:BX:AX zurückgegeben.

- Single-, Double-, Extended- und Comp-Typen (des 80x87-Prozessors) werden auf dem Koprozessor-Stack in ST(0) zurückgegeben.

- Zeiger-Typen werden in DX:AX zurückgegeben.

- String-Typen werden zwischengespeichert. Das *@Result*-Symbol liefert einen Zeiger auf den zwischengespeicherten String.

Die **assembler**-Anweisung ist durchaus mit der **external**-Anweisung vergleichbar – beide Anweisungen gehorchen den selben Richtlinien. Mit den nachfolgenden Beispielen sollen einige Unterschiede zwischen **asm**-Anweisungen in normalen Funktionen und **assembler**-Funktionen verdeutlicht werden. Bei beiden Beispielen wird ein String in Großbuchstaben konvertiert, im ersten Beispiel wird dafür eine **asm**-Anweisung eingesetzt. Beachten Sie, daß der Werte-Parameter *St* in diesem Fall auf eine lokale Variable verweist, denn der Compiler erzeugt automatisch einen Eintritts-Code für die Funktion, der den aktuellen Parameter in den lokalen Speicherbereich kopiert:

```
function UpperCase (Str: String): String;
begin
  asm
    CLD
    LEA     SI,Str
    LES     DI,@Result
    SEGSS   LODSB
```

```
    STOSB
    XOR     AH,AH
    XCHG    AX,CX
    JCXZ    @3
  @1:
    SEGSS   LODSB
    CMP     AL,'a'
    JB      @2
    CMP     AL,'z'
    JA      @2
    SUB     AL,20H
  @2:
    STOSB
    LOOP    @1
  @3:
  end;
end;
```

Das zweite Beispiel stellt die **assembler**-Version von *UpperCase* dar. Hier wird *Str* nicht in den lokalen Speicherbereich kopiert, und die Funktion muß *Str* als **var**-Parameter behandeln:

```
function UpperCase (Str: String): String; assembler
asm
  PUSH    DS
  CLD
  LDS     SI,Str
  LES     DI,@Result
  LODSB
  STOSB
  XOR     AH,AH
  XCHG    AX,CX
  JCXZ    @3
@1:
  LODSB
  CMP     AL,'a'
  JB      @2
  CMP     AL,'z'
  JA      @2
  SUB     AL,20H
@2:
  STOSB
  LOOP    @1
@3:
  POP     DS
end;
```

23. Einbinden von Assembler-Code

Turbo-Pascal-Prozeduren und -Funktionen, die vollständig in Maschinensprache geschrieben sind, können über den Compiler-Befehl {$L *Dateiname*} in Pascal-Programme oder Units aufgenommen werden, wenn sie mit einem Assembler in eine Object-Datei (Suffix .OBJ) übersetzt worden sind. Die Anzahl der {$L}-Befehle pro Programm oder Unit ist nicht begrenzt, für jede Object-Datei ist aber ein eigener Befehl erforderlich.

In Maschinensprache geschriebene Prozeduren und Funktionen müssen innerhalb des Pascal-Quelltextes als **external** deklariert werden. Ein Beispiel:

```
function LoCase(Ch: Char): Char; external;
```

Der Assembler-Quelltext muß alle von Pascal aus verfügbaren Routinen in einem Segment mit einem der Namen CODE, CSEG oder X_TEXT deklarieren, wobei das X in dem Segment-Namen X_TEXT für einen beliebigen Bezeichner steht (zum Beispiel MODUL1_TEXT, OVR_TEXT oder einfach nur _TEXT) Bezeichner müssen darüber hinaus als PUBLIC definiert sein, wenn von Pascal aus darauf zugegriffen wird.

Die Prüfmechanismen von Turbo Pascal greifen bei Maschinenprogrammen nur in sehr begrenztem Rahmen - der Programmierer ist hier selbst dafür verantwortlich, daß die Arbeitsweise der Routine mit ihrer Pascal-Deklaration übereinstimmt (Zahl, Typ und Reihenfolge der Parameter, Codierung als *near* oder *far*, Ergebnistyp bei Funktionen).

Ein Maschinenprogramm kann initialisierte Variablen in einem Segment mit dem Namen CONST oder X_DATA (mit beliebigem Bezeichner X) halten. Uninitialisierte Variable müssen in einem Segment DATA, DSEG oder X_BSS (X beliebig) abgelegt sein.

Alle Bezeichner des Pascal-Programms oder der Unit stehen dem Maschinenprogramm zur Verfügung, solange der Assembler-Quelltext entsprechende EXTRN-Direktiven enthält. (Auch hier liegt es wieder in der Verantwortung des Programmierers, für eine Übereinstimmung der entsprechenden Angaben zu sorgen.)

Turbo Pascal liest mit {$L} angegebene .OBJ-Dateien und nimmt dabei eine Konvertierung des Intel-Formats in das interne Codeformat des Compilers vor. Diese Konvertierung kann nur dann erfolgreich ausgeführt werden, wenn der Programmierer die folgenden Regeln beachtet:

- Alle Prozeduren und Funktionen müssen in einem einzigen Segment stehen, das den Namen CODE, CSEG oder X_CODE trägt, alle initialisierten Variablen müssen in einem einzigen Segment des Namens CONST oder X_DATA stehen, unitialisierte Variablen in DATA, DSEG oder X_BSS. Sämtliche anderen Segment- und/oder GROUP-Direktiven werden ignoriert. Das Code-Segment ist immer BYTE-ausgerichtet, das Daten-Segment beginnt immer an einer WORD-Adresse. Die Deklaration von Segmenten als PUBLIC sowie die Deklaration von Klassennamen (»CLASS«) ist erlaubt und wird von Turbo Pascal ignoriert.

- Für die Angabe von Variablenwerten im Segment DATA, DSEG bzw. X_BSS dürfen Sie nur Fragezeichen verwenden, z.B.

```
Count DW      ?
Buffer        DB      128 DUP (?)
```

- Turbo Pascal ignoriert die Werte aller initialisierten Segmente, außer die vom Code-Segment (CODE, CSEG bzw. X_CODE) und vom initialisierten Daten-Segment (CONST bzw. X_DATA).

- Byte-orientierte Bezugnahmen auf EXTRN-Symbole sind nicht möglich – die Direktiven HIGH und LOW sind deshalb nicht auf Pascal-Bezeichner anwendbar.

Maschinenprogramme mit dem *Turbo Assembler*

Der *Turbo Assembler* (TASM) verfügt über eine Reihe von Pseudobefehlen, die die Programmierung von Maschinensprache-Routinen für Turbo Pascal-Programme stark vereinfachen. Eine einzige Anweisung – nämlich *.MODEL TPASCAL* – setzt die Übergabe-Konventionen von Turbo Pascal, definiert die entsprechenden Segmentnamen und erzeugt den Start- bzw. Endcode für Routinen.

Mit dem Pseudobefehl *PROC* lassen sich die Parameter einer Maschinenroutine in derselben Reihenfolge wie in einem Pascal-Programm deklarieren; RETURNS stellt einen temporären Stringzeiger auf dem Stack zur Verfügung (dessen Platzbedarf automatisch beim RET xx-Befehl berücksichtigt wird).

Ein kurzer Ausblick:

```
        .MODEL TPASCAL
        .CODE
; ------------------------------------------------
; function MyFunc(i,j: Char): string;  { external }
; ------------------------------------------------
MyFunc  PROC FAR i:Byte, j:Byte  RETURNS Result: Word
        PUBLIC MyFunc
                          ;  Startcode wird automatisch eingesetzt
        les di,result     ; Ergebnis-Adresse - ES:DI
        mov al,i          ; Parameter i - AL
```

```
        mov bl,j          ; Parameter j - BL
        ....
        ret               ;  Endcode wird automatisch eingesetzt
```

Weitere Details finden Sie in Kapitel 6 des *Turbo Assembler Benutzerhandbuchs.*

Einige Beispiele

Die folgende Unit enthält zwei in (»konventioneller«, d.h. MASM-kompatibler) Maschinensprache geschriebene Routinen: die Funktion *UpperCase*, die Kleinbuchstaben innerhalb eines Strings in Großbuchstaben verwandelt, und die Funktion *StringOf*, die einen mit vorgegebenen Zeichen gefüllten String bestimmter Länge zurückliefert. Fangen wir mit der Pascal-Deklaration an:

```
unit Strings;

interface

function UpperCase(var S: string): String;
function StringOf(Ch: Char; Count: Byte): String;

implementation

{$L STRS}

function UpperCase; external;
function StringOf; external;

end.
```

Das Maschinenprogramm, in dem die Implementation der Funktionen *UpperCase* und *StringOf* enthalten ist, muß in unserem Beispiel in eine Object-Datei des Namens STRS.OBJ übersetzt werden. Falls Sie einen anderen Namen verwenden wollen, müssen Sie den {$L}-Befehl in der Unit *Strings* entsprechend verändern. Beide Routinen sind als *far* deklariert, weil sie im Inter*face-Teil der Unit erscheinen:*

```
CODE    SEGMENT BYTE PUBLIC
        ASSUME  CS:CODE
        PUBLIC  UpperCase,StringOf   ; für Pascal verfügbar
; ----------------------------------------
; function UpperCase(var S: string): string
; ----------------------------------------
UpperRes        EQU     DWORD PTR [BP+10]
UpperStr        EQU     DWORD PTR [BP+6]
UpperCase       PROC FAR
        PUSH    BP                  ; BP sichern
        MOV     BP,SP               ; Stackzeiger - BP
        PUSH    DS                  ; DS-Register sichern
        LDS     SI,UpperStr         ; Start überg. String - DS:SI
```

```
        LES     DI,UpperRes         ; Start Ergebnis - ES:DI
        CLD                         ; Zählung aufwärts
        LODSB                       ; das ist die Stringlänge
        STOSB                       ; - als 1.Byte Ergebnis
        MOV     CL,AL               ; Stringlänge - CX (Zähler)
        XOR     CH,CH
        JCXZ    U3                  ; Falls 0 Zeichen - Ende
U1:     LODSB                       ; sonst 1 Zeichen laden
        CMP     AL,'a'              ; kleiner 'a' ?
        JB      U2                  ; - ja, keine Veränderung
        CMP     AL,'z'              ; größer 'z' ?
        JA      U11                 ; - ja, keine Veränderung
        SUB     AL,'a'-'A'          ; sonst "Übersetzung"
U2:     STOSB                       ; Zeichen als Ergebnis speichern
        LOOP    U1                  ; nächster Durchlauf
U3:     POP     DS                  ; DS zurück
        POP     BP                  ; BP zurück
        RET     4                   ; Parameter vom Stack & Ende
UpperCase       ENDP
```

Hinweis: UpperCase erzeugt keine temporäre Kopie des übergebenen Parameters *S*, der aus diesem Grund als **var** deklariert sein muß.

```
; ------------------------------------------------
; function StringOf(Ch: Char; Count: Byte): string
; ------------------------------------------------
StrOfRes        EQU     DWORD PTR [BP+10]
StrOfChar       EQU     BYTE PTR [BP+8]
StrOfCount      EQU     BYTE PTR [BP+6]
StringOf        PROC FAR
        PUSH    BP                  ; BP sichern
        MOV     BP,SP               ; Stackzeiger - BP
        LES     DI,StrOfRes         ; Startadresse Ergebnis - ES:DI
        MOV     AL,StrOfCount       ; Anzahl (Parameter)
        CLD                         ; Zählung aufwärts
        STOSB                       ; Längenbyte ins Ergebnis
        MOV     CL,AL               ; und CX als Zähler
        XOR     CH,CH
        MOV     AL,StrOfChar        ; zu wiederholendes Zeichen (Parm)
        REP     STOSB               ; CX Zeichen ins Ergebnis
        POP                         ; BP zurück
        RET     4                   ; Parameter vom Stack & Ende
StringOf        ENDP

CODE    ENDS
        END
```

STRS.ASM und die Unit *Strings* werden mit den folgenden Befehlen (auf der DOS-Kommandoebene) übersetzt:

```
TASM STRS        übersetzt STRS.ASM und erzeugt STRS.OBJ
TPCW STRINGS     übersetzt STRINGS.PAS, bindet STRS.OBJ ein und
                   erzeugt STRINGS.TPU
```

Das nächste Beispiel zeigt, wie Maschinenprogramme auf Routinen und Variablen von Pascal zugreifen können: Das Programm *Numbers* liest maximal 100 Integerzahlen ein und ruft danach die in Maschinensprache geschriebene Prozedur *CheckRange* auf, die jeden dieser Werte überprüft. Wenn ein Wert außerhalb des vorgegebenen Bereichs liegt, dann ruft *CheckRange* die Pascal-Prozedur *RangeError* zur Ausgabe einer entsprechenden Meldung auf, wobei der entsprechende Wert als Parameter übergeben wird.

```
program Numbers;
uses WinCrt;
{$L CHECK}
var
  Buffer: array[1..100] of Integer;
  Count: Integer;
procedure RangeError(N: Integer);
begin
  Writeln(N:6, ' liegt außerhalb des Bereichs');
end;

procedure CheckRange(Min,Max: Integer); external;

begin   { Hauptprogramm }
  Count := 0;
  while not Eof and (Count) do   { hier zu Testzwecken nach fünf }
  begin                            { Werten beendet }
    Inc(Count);
    Readln(Data[Count]);
  end;
  CheckRange(-10,10);  { Überprüfung der Werte }
end.
```

Wir haben hier den Namen CHECK zusammen mit dem {$L}-Befehl angegeben: der folgende Assembler-Quelltext muß also in eine Datei des Namens CHECK.OBJ übersetzt werden. Im Gegensatz zum vorangegangenen Beispiel ist die Maschinensprache-Prozedur *CheckRange* innerhalb eines Programms deklariert und wird deshalb als **near** aufgerufen.

```
DATA    SEGMENT WORD PUBLIC
        EXTRN   Data:WORD,Count:WORD    ; Pascal-Variablen
DATA    ENDS
CODE    SEGMENT BYTE PUBLIC
        ASSUME  CS:CODE,DS:DATA
        EXTRN   RangeError:NEAR  ; Pascal-Routine
        PUBLIC  CheckRange       ; Maschinenroutine

CheckRange      PROC    NEAR
        MOV     BX,SP            ; Zeiger auf die Parameter - BX
```

```
        MOV     AX,SS:[BX+4]     ; Parameter Min - AX
        MOV     DX,SS:[BX+2]     ; Parameter Max - DX
        XOR     BX,BX            ; BX auf 0
        MOV     CX,Count         ; Anzahl
        JCXZ    SD4              ; Ende wenn 0
SD1:    CMP     Buffer[BX],AX    ; zu klein?
        JL      SD2              ; - ja
        CMP     Buffer[BX],DX    ; zu groß?
        JLE     SD3              ; - nein
SD2:    PUSH    AX               ; Überschreitung: Register sichern
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    Buffer[BX]       ; der unzulässige Wert als Parameter
        CALL    RangeError       ; Pascal-Aufruf
        POP     DX               ; Register zurück
        POP     CX
        POP     BX
        POP     AX
SD3:    INC     BX               ; nächstes Element
        INC     BX
        LOOP    SD1              ; nächster Durchlauf
SD4:    RET     4                ; Stack aufräumen & Ende
CheckRange      ENDP

CODE    ENDS
        END
```

Ein weiterer Ausblick

Der folgende – alternative – Quelltext für die Prozedur *CheckRange* nutzt die Möglichkeiten des *Turbo Assemblers* und demonstriert dabei, wieviele Formalitäten sich mit diesem Programm einsparen lassen:

```
        .MODEL TPASCAL           ; Konventionen von Turbo Pascal
        LOCALS @@                ; Präfix für lokale Symbole: @@
        .DATA
        EXTRN   Data:WORD,Count:WORD    ; Pascal-Variablen
        .CODE
        EXTRN   RangeError:NEAR  ; Pascal-Routine
        PUBLIC  CheckRange       ; Maschinenroutine

CheckRange  PROC NEAR Min:WORD, Max:WORD
        MOV     AX,Min           ; Parameter Min - AX
        MOV     DX,Max           ; Parameter Max - DX
        XOR     BX,BX            ; BX auf 0
        MOV     CX,Count         ; Anzahl
```

```
        JCXZ    @@4             ; Ende wenn 0
@@1:    CMP     Data[BX],AX     ; zu klein?
        JL      @@2             ; - ja
        CMP     Data[BX],DX     ; zu groß?
        JLE     @@3             ; - nein
@@2:    PUSH    AX              ; Überschreitung: Register sichern
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    Data[BX]        ; der unzulässige Wert als Parameter
        CALL    RangeError      ; Pascal-Aufruf
        POP     DX              ; Register zurück
        POP     CX
        POP     BX
        POP     AX
@@3:    INC     BX              ; nächstes Element
        INC     BX
        LOOP    @@1             ; nächster Durchlauf
@@4:    RET                     ; Stack aufräumen & Ende
CheckRange      ENDP
        END
```

Wie bereits im letzten Abschnitt erläutert, kümmert sich der *Turbo Assembler* nach dem Befehl *.MODEL TPASCAL* automatisch um die Deklaration von Segmenten sowie den Start- und Endcode der Routinen.

Maschinencode mit inline

Für sehr kurze Unterprogramme in Maschinensprache sind in Turbo Pascal keine eigenen Object-Dateien notwendig: über **inline** lassen sich einzelne Bytefolgen direkt in den Pascal-Quelltext eines Programms einfügen.

inline-Anweisungen

bestehen aus dem reservierten Wort **inline**, gefolgt von einem Klammerpaar mit einem oder mehreren Elementen, die untereinander durch Schrägstriche voneinander getrennt sind - so, wie es das folgende kurze Beispiel für diesen langen Satz demonstriert:

```
inline(10/$2345/Count+1/Data-Offset);
```

Formal gesehen hat **inline** die Syntax:

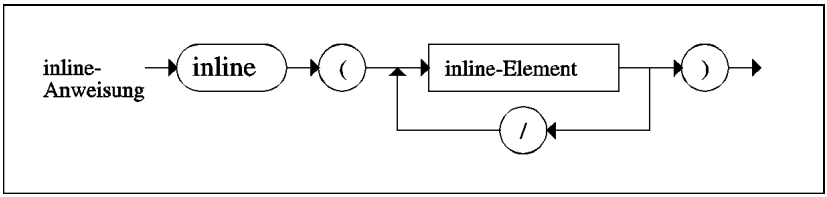


Abbildung 23.1 Syntax von inline

Jedes Element besteht aus einer optionalen Größenangabe (oder) und einer Konstanten oder einem Variablenbezeichner. Über ein nachfolgendes + oder - zusammen mit einer Konstanten sind relative Angaben möglich:

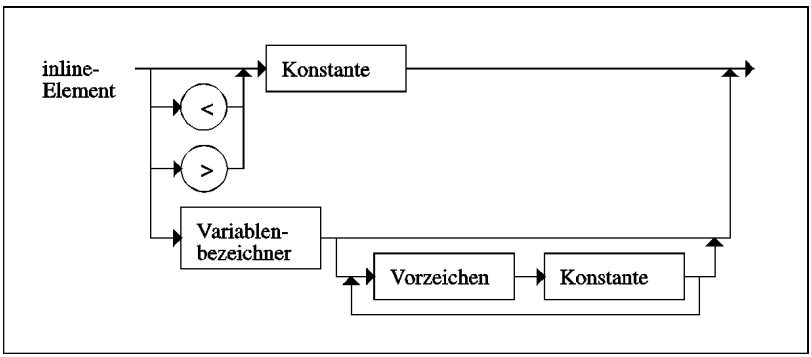


Abbildung 23.2 Syntax eines inline-Elements

Der Wert jedes **inline**-Elements ergibt sich aus den angegebenen Konstanten und/oder den Adressen der angegebenen Bezeichner. Wenn sich die Angabe ausschließlich aus Konstanten zusammensetzt und das Gesamtergebnis im Bereich von 0..255 liegt, dann erzeugt **inline** ein einzelnes Byte. Ergebnisse außerhalb dieses Bereichs oder Angaben von Variablennamen erzeugen grundsätzlich ein Wort (2 Bytes), dessen niederwertiger Teil auf der niedrigeren Adresse gespeichert wird.

Über die Operatoren läßt sich diese automatische Größenbestimmung außer Kraft setzen: **inline**-Elemente, die mit eingeleitet werden, speichern nur den niederwertigen Teil ihres Ergebnisses mit einem Byte, die Einleitung mit erzeugt immer ein Wort (dessen höherwertiger Teil unter Umständen den Wert 0 hat). Der Befehl

```
inline(1234/$44);
```

erzeugt also drei Bytes Code, nämlich $34 (den niederwertigen Teil von $1234), $44 und $00.

Bei der Angabe eines Variablennamens erzeugt **inline** nicht den (zur Compilierungszeit sowieso unbekannten) Wert dieser Variablen, sondern eine Adres-

se mit 16 Bit, die sich auf das jeweilige Segment bezieht. Globale Variablen (die außerhalb aller Prozeduren und Funktionen bzw. im Interface-Teil einer Unit deklariert sind), werden vom Compiler im Datensegment gespeichert, auf das über das Register DS zugegriffen werden kann. Wenn wir annehmen, daß die Variable *Zaehler* ab der Adresse $4628 im Daten-Segment steht, dann erzeugt eine Angabe wie

```
inline(Zaehler);
```

den Wert $4628, der mit zwei aufeinanderfolgenden Bytes ($28, $46) gespeichert wird.

Lokale Variablen (die innerhalb einer Prozedur oder Funktion deklariert sind) werden im Stack-Segment gespeichert. Hier berechnet **inline** nicht eine »absolute« Adresse, sondern erzeugt automatisch einen Relativwert zum Inhalt des BP-Registers.

Hinweis: Die Register BP, SP, SS und DS dürfen durch **inline**-Anweisungen nicht verändert werden – alle anderen Register stehen dem Programmierer frei zur Verfügung.

Die im folgenden Beispiel gezeigte Prozedur *FillWord* erwartet drei Parameter: einen Zeiger mit 32 Bit auf den zu füllenden Speicherbereich (*Ziel*), die Größe dieses Bereichs in (16 Bit-) Worten (*Zahl*) und den Wert, mit dem der Bereich gefüllt werden soll (*Datum*):

```
procedure FillWord(var Ziel ^Word; var Zahl, Datum: Word);
begin
  inline (
    $C4/$BE/Ziel/     { LES   DI,Ziel[BP]  }
    $8B/$8E/Zahl/     { MOV   CX,Zahl[BP]  }
    $8B/$86/Datum/    { MOV   AX,Datum[BP] }
    $FC/              { CLD                }
    $F3/$AB);         { REP   STOSW        }
end;
```

inline-Anweisungen können frei mit anderen Pascal-Befehlen innerhalb des jeweiligen Blocks gemischt werden.

inline-Deklarationen

haben dieselbe Syntax wie **inline**-Anweisungen und im Prinzip denselben Zweck;

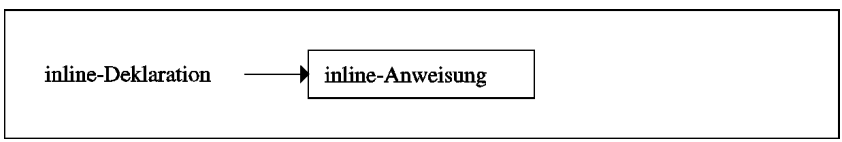


Abbildung 23.3 Syntax einer inline-Deklaration

Für normale Prozeduren und Funktionen - unabhängig davon, ob sie **inline**-Anweisungen enthalten oder nicht – erzeugt der Compiler Code, durch den die entsprechenden Parameter auf den Stack gebracht werden, und hängt schließlich einen CALL-Befehl für den eigentlichen Aufruf an. Als **inline** deklarierte Routinen werden dagegen wie Assembler-Makros behandelt; anstatt sie über einen CALL-Befehl aufzurufen, wiederholt der Compiler einfach die mit **inline** angegebenen Befehle jedesmal, wenn der Bezeichner dieser Routine im Quelltext erscheint. Ein Beispiel:

```
procedure Interrupts_Aus;
inline($FA);    { CLI }

procedure Interrupts_Ein
begin
  inline($FB);  { STI }
end;
```

Interrupts_Ein ist eine völlig normale Prozedur, die mit Befehlen zur Sicherung der Register beginnt, danach den eigentlichen Code ausführt und vor dem Rücksprung die Register wiederherstellt. Der Compiler erzeugt für jedes Vorkommen des Bezeichners *Interrupts_Ein* einen CALL-Befehl. Anders dagegen bei *Interrupts_Aus*: hier fügt der Compiler anstelle eines Aufrufs einfach das Byte $FA in den laufenden Code ein.

Als **inline** deklarierte Prozeduren und Funktionen arbeiten ohne Start- und Endcode; da sie jeweils direkt in das Programm eingefügt werden, dürfen sie keine Rücksprungbefehle enthalten. Die Übergabe von Parametern ist erlaubt, sie werden vom Compiler wie bei einem normalen Aufruf auf den Stack gelegt und müssen innerhalb der Routine auch wieder entfernt werden. Was dagegen nicht funktioniert, ist eine symbolische Bezugname über die Parameternamen: ohne Startcode ist keine Adressierung über das BP-Register möglich. Ein weiteres Beispiel:

```
function LongMult(X, Y: Integer): LongInt;
inline(
  $58/          { POP   DX    ; Parameter X }
  $5A/          { POP   AX    ; Parameter Y }
  $F7/$EA);     { IMUL  DX    ; X * Y - DX:AX }
```

Für jeden Aufruf der Form

```
Z := LongMult(A,B);
```

erzeugt der Compiler Befehle, mit denen die Werte von *A* und *B* auf den Stack gelegt werden, fügt dann direkt die vier Bytes $58/$5A/$F7/$EA in den laufenden Code ein und hängt schließlich Befehle an, die den Inhalt des Registerpaars DX:AX in der Variablen *Z* speichern. Weitere Befehle (spezifisch: Start- und Endcode einer Prozedur oder Funktion) werden nicht erzeugt.

inline-Deklarationen sind nur für sehr kurze Routinen (ca. 10 Bytes) gedacht. Da der Compiler ihren Code jedesmal direkt einfügt, haben sie im Gegensatz zu anderen Bezeichnern keine feste Adresse und können nicht als Argumente der Funktionen *Addr*, *Ofs* und *Seg* oder als Operand von *@* verwendet werden.

V. Die Turbo Pascal Referenz

24. Verzeichnis der Routinen

Die folgenden Seiten enthalten eine alphabetische Auflistung der Prozeduren und Funktionen von Turbo Pascal für Windows zusammen mit den Routinen der Standard-Units *WinDos*, *WinCrt* und *Strings*.

Alle Einträge dieses Verzeichnisses haben dieselbe Form – je nach Art der Routine kann allerdings der eine oder andere Unterpunkt fehlen:

Beispiel-Prozedur **Unitname**

Funktion	was die Routine bewirkt.
Deklaration	die formale Deklaration, benutzerdefinierte Elemente sind *kursiv* gesetzt.
Ergebnistyp	der Typ des zurückgelieferten Wertes (nur bei Funktionen).
Erläuterung	generelle Informationen zur Arbeitsweise.
Grenzen	Einschränkungen und Hinweise auf mögliche Probleme.
Querverweis	auf Routinen ähnlicher, verwandter oder entgegengesetzter Arbeitsweise.
Beispiel	Ein kurzes Programm oder Programmfragment, das diese Routine benutzt. Alle Beispiele, die Ausgaben erzeugen (Writeln) oder Eingaben erfordern (Readln), sollten die Unit WinCrt verwenden.

Abs-Funktion

Funktion	liefert den absoluten Wert des Arguments zurück.
Deklaration	`Abs(x)`
Ergebnistyp	derselbe wie der Typ des übergebenen Arguments.
Erläuterung	*x* ist ein Ausdruck mit einem Integer- oder Real-Typ. Das Ergebnis hat denselben Typ und stellt den absoluten Wert von *x* dar.

Beispiel
```
var
  r: Real;
  x: Integer;
begin
  r := Abs(-2.3);    { r := 2.3; }
  x := Abs(-157);    { x := 157; }
end.
```

Addr-Funktion

Funktion liefert die Adresse des angegebenen Objekts zurück.

Deklaration `Addr(x)`

Ergebnistyp *Pointer*

Erläuterung *x* ist eine beliebige Variable oder der Bezeichner einer Prozedur oder Funktion, die nicht als **inline** deklariert ist. Das Ergebnis ist ein Zeiger, der auf *x* zeigt (also die Adresse von *x* enthält). Das Ergebnis von *Addr* ist wie **nil** zuweisungskompatibel zu allen Zeiger-Typen.

Hinweis: Der Adreß-Operator *@* hat dasselbe Ergebnis wie die Funktion *Addr*.

Querverweis *Ofs*, *Ptr*, *Seg*

Beispiel
```
var
  p : Pointer;
begin
  p := Addr(p);  { p zeigt jetzt auf sich selbst }
end.
```

Append-Prozedur

Funktion öffnet eine existierende Datei für das Anhängen weiterer Daten.

Deklaration `Append(var f: Text)`

Erläuterung *f* ist eine Textdatei-Variable, die zuvor über *Assign* mit einer externen Datei verbunden worden sein muß.

Append öffnet die Datei und setzt die momentane Position innerhalb dieser Datei an ihr Ende, folgende Schreibaktionen hängen also neue Daten an. Falls die Datei bereits offen war, wird sie durch *Append* zuerst geschlossen und dann erneut geöffnet.

Wenn der letzte 128-Byte-Block der Datei ein ^Z (ASCII-Code 26) enthält, setzt *Append* die momentane Position innerhalb der Datei so, daß die nächste Schreibaktion dieses Zeichen

überschreibt. Auf diese Weise läßt sich Text auch an Dateien anhängen, die mit ^Z abgeschlossen wurden.

Wenn *f* mit *Assign(f,'')* ein Nullstring als Name zugeordnet wurde, dann bezieht sich diese Datei-Variable nach dem Aufruf von *Append* auf die Standard-Ausgabe (Standard-Handle 1, entspricht *Output*).

In eine mit *Append* geöffnete Datei kann nur geschrieben werden - jeder Aufruf der Form *Eof(f)* liefert *True* als Ergebnis.

Im Modus {$I-} liefert die Funktion *IOResult* den Wert 0 zurück, wenn *Append* fehlerfrei durchgeführt werden konnte, ansonsten liefert sie einen Fehlercode <> 0.

Grenzen neue Dateien werden von *Append* nicht erzeugt – die mit *f* verbundene Datei muß also bereits existieren.

Querverweis *Assign*, *Close*, *Reset*, *Rewrite*

Beispiel
```
var f: Text;
begin
  Assign(f, 'TEST.TXT');  { Zuordnung des Namens }
  Rewrite(f);     { erzeugt eine neue (leere) Datei }
  Writeln(f, 'Eine Zeile');
  Close(f);
  Append(f);      { öffnet die Datei erneut }
  Writeln(f, 'Diese Zeile wird angehängt!');
  Close(f);     { Datei enthält jetzt zwei Textzeilen }
end.
```

ArcTan-Funktion

Funktion liefert den Arcustangens des Arguments zurück.

Deklaration `ArcTan(x: Real)`

Ergebnistyp *Real*

Erläuterung *x* ist ein Ausdruck mit einem Real-Typ. Das Funktionsergebnis ist der Arcustangens von *x* (also der Wert, der als Argument einer Tangens-Berechnung den Wert *x* ergeben würde). Die zugrundeliegende Einheit ist *Rad*.

Querverweis *Cos*, *Sin*

Beispiel
```
var
  r : Real;
begin
  r := ArcTan(Pi);
end.
```

Assign-Prozedur

Funktion ordnet einer Datei-Variablen eine externe Datei zu.

Deklaration
```
Assign(var f; Name: string)
```

Erläuterung *f* ist eine Datei-Variable (die für eine typisierte, untypisierte oder eine Text-Datei stehen kann), *Name* ist ein String-Ausdruck.

Assign ordnet *f* die durch *Name* angegebene Datei zu. Alle weiteren Operationen mit *f* beziehen sich solange auf diese Datei, bis *f* durch einen weiteren Aufruf von *Assign* eine andere externe Datei zugeordnet wird.

Ein Dateiname besteht aus einem optionalen Laufwerksbezeichner sowie einer optionalen Anzahl von Verzeichnisnamen, die jeweils durch einen umgekehrten Schrägstrich (»backslash«) voneinander getrennt sind. Den letzten Teil der Angabe bildet der Name der Datei selbst:

```
Laufwerk:\Dir_Name\...\Dir_Name\Dateiname
```

Wenn die Angabe nach *Laufwerk* mit einem umgekehrten Schrägstrich beginnt, wird sie auf das Stammverzeichnis des jeweiligen Laufwerks bezogen, ansonsten auf das momentan gesetzte Verzeichnis. Wenn *Laufwerk* nicht angegeben ist, wird das momentan gesetzte Laufwerk verwendet. *Dir_Name* und *Dateiname* bestehen jeweils aus maximal 8 Zeichen, *Dateiname* kann ein zusätzliches *Suffix* enthalten (drei Zeichen, abgetrennt durch einen Punkt).

Ein vollständiger Dateiname (Laufwerk, Suchweg und Dateiname selbst) darf maximal 79 Zeichen umfassen.

Wenn *f* einen Nullstring darstellt, dann ordnen *Assign* und ein nachfolgendes *Reset* oder *Rewrite* dieser Datei-Variablen die Standard-Datei *Input* bzw. *Output* zu - auf diese Weise sind Umleitungen der Ein-/Ausgabe über DOS möglich.

Grenzen *Assign* darf nie auf eine offene Datei angewendet werden.

Querverweis *Append*, *Close*, *Reset*, *Rewrite*

Beispiel
```
{ Die Ausgabe dieses Programms läßt sich mit DOS
  beliebig auf einen Drucker, in eine Datei
  etc. umleiten }
var
  f : Text;
begin
  Assign(f, '');   { Standard-Datei }
  Rewrite(f);     { setzt f als Output }
  Writeln(f, 'Diese Ausgabe läßt sich umleiten!');
  Close(f);
end.
```

AssignCrt-Prozedur **WinCrt**

Funktion ordnet dem Bildschirm (»CRT«) eine Textdatei zu.

Deklaration
```
AssignCrt(var f: Text)
```

Erläuterung *AssignCrt* arbeitet exakt so wie die Standardprozedur *Assign*. Der einzige Unterschied liegt darin, daß kein Dateiname angegeben wird: *AssignCrt* verbindet die Datei-Variable automatisch mit dem Bildschirm.

Danach erfolgende *Write*- und *WriteLn*-Operationen schreiben in das CRT-Fenster, *Read*- und *ReadLn*-Operationen lesen aus dem CRT-Fenster.

BlockRead-Prozedur

Funktion liest ein oder mehrere Records einer Datei in eine Puffervariable.

Deklaration
```
BlockRead(var f: File; var buf;
          count: Word [; result: Word])
```

Erläuterung *f* steht für eine untypisierte Datei-Variable, *buf* für eine Variable beliebigen Typs (normalerweise ein *Byte*- oder *Char*-Array); *count* ist ein Ausdruck des Typs *Word*, *result* hat ebenfalls den Typ *Word* und muß eine Variable sein.

BlockRead liest *count* Records der durch *f* angegebenen Datei und speichert sie ab der Startadresse von *buf*. Gelesen wird ab der momentanen Position innerhalb von *f*. Wenn *result* als Parameter übergeben wurde, dann beendet *BlockRead* den Lesevorgang bei Erreichen des Dateiendes ohne Fehlermeldung und setzt *result* auf die Anzahl der gelesenen Records (*result* erhält also maximal den Wert von *count*); ist *result* dagegen nicht übergeben worden, dann erzeugt *BlockRead* bei Überschreitung des Dateiendes einen Laufzeitfehler.

Hinweis: result gibt die Anzahl der *komplett* gelesenen Records wieder. Wenn von einer Datei mit 128 Bytes Recordgröße 255 Bytes vor dem Erreichen des Dateiendes gelesen werden konnten, dann hat *result* also den Wert 1.

Die Gesamtzahl der zu übertragenden Bytes ergibt sich aus *count* * *RecSize*, wobei *RecSize* die bei der Eröffnung der Datei definierte Recordgröße ist (und von Turbo Pascal automatisch auf 128 gesetzt wird, wenn keine explizite Angabe erfolgt). Maximal können mit einem Aufruf von *BlockRead* 65535 Bytes (64 KByte) gelesen werden.

Die momentane Position innerhalb der Datei wird von *BlockRead* um *result* Records in Richtung auf das Dateiende versetzt.

Wenn *BlockRead* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* (im Modus {$I-}) den Wert 0; ansonsten hat diese Funktion einen Fehlercode <> 0 als Ergebnis.

Grenzen die durch *f* angegebene Datei muß offen sein.

Querverweis *BlockWrite*

Beispiel

```
program Copy;
{ ein einfaches Kopierprogramm OHNE jede Fehlerprüfung }
var
  Quelle, Ziel: file;
  R_gelesen, R_geschrieben: Word;
  Puffer: array[1..2048] of Byte;

begin
  { Zuordnung und Öffnen der Quelldatei }
  Assign(Quelle, ParamStr(1));
  Reset(Quelle, 1);  { Recordlänge: 1 Byte }
  { Zuordnung und Erzeugen der Zieldatei }
  Assign(Ziel, ParamStr(2)); Rewrite(Ziel, 1);
  Writeln(FileSize(Quelle), ' Bytes werden kopiert...');
  repeat
    BlockRead
    (Quelle, Puffer, SizeOf(Puffer), R_gelesen);
    BlockWrite(Ziel, Puffer, R_gelesen, R_geschrieben);
  until (R_gelesen = 0) or (R_gelesen <> R_geschrieben);
  Close(Quelle); Close(Ziel);
end.
```

BlockWrite-Prozedur

Funktion schreibt ein oder mehrere Records aus einer Puffervariablen in eine Datei.

Deklaration

```
BlockWrite(var f: File; var buf;
           count: Word [; result: Word])
```

Erläuterung *f* steht für eine untypisierte Datei-Variable, *buf* für eine Variable beliebigen Typs (normalerweise ein *Byte*- oder *Char*-Array), *count* ist ein Ausdruck des Typs *Word*, *result* hat ebenfalls den Typ *Word* und muß eine Variable sein.

BlockWrite liest *count* Records ab der Startadresse von *buf* aus dem Speicher und schreibt sie in die durch *f* angegebene Datei. Geschrieben wird ab der momentanen Position innerhalb von *f*. Wenn *result* als Parameter übergeben wurde, dann beendet *BlockWrite* den Schreibvorgang auch dann ohne Fehlermeldung und setzt *result* auf die Anzahl der geschriebenen Records (*result* erhält also maximal den Wert von *count*), wenn aus

irgendeinem Grund (am wahrscheinlichsten: Diskette voll) nicht sämtliche Records geschrieben werden konnten; ist *result* dagegen nicht übergeben worden, dann erzeugt *BlockWrite* in solchen Fällen einen Laufzeitfehler.

Hinweis: result gibt die Anzahl der *komplett* geschriebenen Records wieder. Wenn in eine Datei mit 128 Bytes Recordgröße 255 Bytes geschrieben werden konnten, dann hat *result* also den Wert 1.

Die Gesamtzahl der zu übertragenden Bytes ergibt sich aus *count* * *RecSize*, wobei *RecSize* die bei der Öffnung der Datei definierte Recordgröße ist (und von Turbo Pascal automatisch auf 128 gesetzt wird, wenn keine explizite Angabe erfolgt). Maximal können mit einem Aufruf von *BlockWrite* 65535 Bytes (64 KByte) geschrieben werden.

Die momentane Position innerhalb der Datei wird von *BlockWrite* um *result* Records in Richtung auf das Dateiende versetzt.

Wenn *BlockWrite* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* (im Modus {$I-}) den Wert 0; ansonsten hat diese Funktion einen Fehlercode <> 0 als Ergebnis.

Grenzen die durch *f* angegebene Datei muß offen sein.

Querverweis *BlockRead*

Beispiel siehe *BlockRead*

ChDir-Prozedur

Funktion wechselt das Standard-Directory.

Deklaration `ChDir(s: string)`

Erläuterung *s* ist ein Stringausdruck und gibt das neu zu setzende Standard-Directory an. Wenn *s* einen Laufwerksbezeichner enthält, wird das Laufwerk ebenfalls gewechselt. Wenn der durch *s* angegebene Suchweg mit einem umgekehrten Schrägstrich (»backslash«) beginnt, bezieht er sich auf das Stammverzeichnis des Laufwerks, ansonsten erfolgt der Wechsel relativ zum momentan gesetzten Verzeichnis.

Wenn *ChDir* fehlerfrei ausgeführt werden konnte, liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten liefert diese Funktion den entsprechenden Fehlercode (<> 0) zurück.

Querverweis *GetDir*, *MkDir*, *RmDir*. *SetCurDir* hat dieselbe Funktion, benötigt aber als Argument einen null-terminierten String und keinen Pascal-String.

Beispiel
```
begin { Directory-Name wird als erster Kommandozeilen-
        Parameter angenommen: }
  {$I-}
  ChDir(ParamStr(1));
  if IOResult <> 0 then
    Writeln('Directory kann nicht gesetzt werden!');
end.
```

Chr-Funktion

Funktion liefert ein Zeichen der angegebenen Ordinalität zurück.

Deklaration `Chr(x: Byte)`

Ergebnistyp *Char*

Erläuterung *x* ist ein Integer-Ausdruck. Das Ergebnis der Funktion ist ein Zeichen, dessen ASCII-Code dem Wert von *x* entspricht.

Querverweis *Ord*

Beispiel
```
uses WinCrt
var
  I: Integer;
begin
  for I := 32 to 255 do Write(Chr(I));
end.
```

Close-Prozedur

Funktion schließt eine Datei.

Deklaration `Close(var f)`

Erläuterung *f* ist eine Datei-Variable beliebigen Typs und mit einer externen Datei verbunden, die zuvor mit *Reset*, *Rewrite* oder *Append* geöffnet wurde. Diese Datei wird auf den neuesten Stand gebracht, über einen Aufruf von *Flush* wird ein Schreiben eventuell noch »ungeschriebener« Daten erzwungen; danach gibt *Close* den DOS-Handle wieder frei.

Hinweis: Die Verbindung zwischen der Datei-Variablen und dem mit *Assign* zugeordneten Namen wird durch *Close* nicht aufgehoben.

Wenn *Close* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Querverweis *Append*, *Assign*, *Reset*, *Rewrite*

Beispiel

```
var f: file;
begin
  Assign(f, '\AUTOEXEC.BAT');
  Reset(f, 1);                { Recordgröße 1 }
  Writeln('AUTOEXEC.BAT ist ',
          FileSize(f), ' Bytes lang.');
  Close(f);
end.
```

ClrEol-Prozedur **WinCrt**

Funktion löscht alle Zeichen von der momentanen Position des Cursors bis zum Zeilenende.

Deklaration `ClrEol`

Querverweis *ClrScr*

ClrScr-Prozedur **WinCrt**

Funktion löscht den Bildschirm;

Deklaration `ClrScr`

Erläuterung Wenn der Bildschirm gelöscht ist, wird der Cursor in die obere linke Ecke gesetzt.

Querverweis *ClrEol*

Concat-Funktion

Funktion verbindet zwei oder mehrere Stringteile miteinander.

Deklaration `Concat(s1 [, s2, ..., sn]: string)`

Ergebnistyp *String*

Erläuterung Jeder Parameter von *Concat* ist ein String-Ausdruck. Das Ergebnis der Funktion ist ein String, der sich aus allen übergebenen Argumenten in der Reihenfolge ihrer Angabe zusammensetzt und automatisch nach dem 255. Zeichen abgeschnitten wird.

Hinweis: In Turbo Pascal hat der Operator + dieselbe Wirkung wie die Standardfunktion *Concat*.

Querverweis *Copy*, *Delete*, *Insert*, *Length*, *Pos*

Beispiel

```
var
  s: string;
begin
  s := Concat('ABC', 'DEF');  { s := 'ABCDEF'}
```

```
  s := 'ABC' + 'DEF'    { hat dasselbe Ergebnis }
end.
```

Copy-Funktion

Funktion	liefert einen Teil eines Strings zurück.
Deklaration	`Copy(s: string; index: Integer, count: Integer)`
Ergebnistyp	*String*
Erläuterung	*Copy* liefert *count* Zeichen ab der durch *index* angegebenen Position des Strings *s* zurück. *s* ist ein String-Ausdruck, *count* und *index* sind Integer-Ausdrücke. Wenn *index* größer als die momentane Länge von *s* ist, liefert *Copy* einen Nullstring; wenn *s* kürzer als *index* + *count* ist, werden nur die Zeichen zurückgeliefert, die *s* ab der durch *index* angegebenen Position enthält.
Querverweis	*Concat*, *Insert*, *Delete*, *Length*, *Pos*

Beispiel	`var`
	`  s : string;`
	`begin`
	`  s := 'ABCDEF';`
	`  Writeln(Copy(s,2,3));  { 'BCD');`
	`  Writeln(Copy(s,100,10));  { '' }`
	`end.`

Cos-Funktion

Funktion	liefert den Cosinus des Arguments zurück.
Deklaration	`Cos(x: Real)`
Ergebnistyp	*Real*
Erläuterung	*x* ist ein Real-Ausdruck und wird als Winkel in der Einheit *Rad* (0..2 Pi = 0..360 Grad) interpretiert. Das Ergebnis der Funktion ist der Cosinus von *x*.
Querverweis	*ArcTan*, *Sin*

Beispiel	`var r: Real;`
	`begin`
	`  r := Cos(Pi);   { Ergebnis -1 }`
	`end.`

CreateDir-Prozedur **WinDos**

Funktion erzeugt ein neues Unterverzeichnis.

Deklaration `CreateDir (Dir: PChar)`

Erläuterung Das zu erstellende Unterverzeichnis wird in *Dir* angegeben. Fehler werden in *DosError* angeben. *MkDir* hat dieselbe Funktion wie *CreateDir*, verwendet jedoch einen Pascal-String als Argument, nicht einen null-terminierten String.

Querverweis *GetCurDir, SetCurDir, RemoveDir*

CSeg-Funktion

Funktion liefert die Adresse des momentanen Code-Segments zurück.

Deklaration `CSeg`

Ergebnistyp *Word*

Erläuterung *CSeg* liefert den momentanen Inhalt des Prozessor-Registers CS zurück - also die Adresse des Segments, in dem der Code des »momentanen« Moduls gespeichert ist.

Querverweis *DSeg*, *SSeg*

CursorTo-Prozedur **WinCrt**

Funktion bewegt den Cursor auf die angegebenen Koordinaten im virtuellen Bildschirm.

Deklaration `CursorTo (X, Y: Integer)`

Erläuterung Die obere linke Ecke entspricht den Koordinaten (0,0). Die Variable *Cursor* speichert die neuen Koordinaten (X,Y).

Dec-Prozedur

Funktion erniedrigt eine Variable.

Deklaration `Dec(x [, n: LongInt])`

Erläuterung *x* ist eine Variable ordinalen Typs, der optionale Parameter *n* ist ein Integer-Ausdruck. Wenn *n* nicht angegeben ist, erniedrigt *Dec* die Variable *x* um 1 (und entspricht damit der Zuweisung `x := x - 1`), ansonsten wird *x* um *n* erniedrigt (`x := x - n`).

Dec ist eine Art **inline**-Prozedur und erzeugt optimierten Code. Die Vorteile dieser »Abkürzung« kommen am meisten in engen Schleifen und bei komplexen Adressierungen zum Tragen.

Querverweis *Inc, Pred, Succ*

Beispiel
```
var
  IntVar : Integer;
  LongVar: LongInt;
begin
  IntVar := 10; LongVar := 90234;
  Dec(IntVar);       { IntVar := IntVar - 1    (= 9) }
  Dec(LongVar,5);    { LongVar := LongVar - 5 (= 90229)}
end.
```

Delete-Prozedur

Funktion löscht einen Teil eines Strings.

Deklaration `Delete(var s: string; index, count: Integer)`

Erläuterung *s* ist eine String-Variable, *index* und *count* sind Integer-Ausdrücke. *Delete* löscht ab der durch *index* angegebenen Position *count* Zeichen aus dem String *s*. Wenn *index* größer als die momentane Länge von *s* ist, passiert gar nichts; wenn *s* kürzer oder genauso lang wie *index* + *count* ist, läßt *Delete* nur die vor der Position *index* stehenden Zeichen übrig.

Querverweis *Concat, Copy, Insert, Length, Pos*

Beispiel
```
var  s: string
begin
  s := 'ABCDEF';
  Delete (s,3,1);  { 'ABDEF' }
  Delete (s,4,99);  { 'ABD' }
end.
```

DiskFree-Funktion **WinDos**

Funktion liefert die Größe des freien Speicherplatzes auf einem Laufwerk zurück.

Deklaration `DiskFree(Drive: Byte)`

Ergebnistyp *LongInt*

Erläuterung *Drive* ist ein Ausdruck des Typs *Byte* und gibt das Laufwerk an, dessen freier Speicherplatz (in Bytes) ermittelt werden soll: der Wert 0 steht für das momentan gesetzte Laufwerk, der Wert 1 für Laufwerk A:, 2 für B: usw. *DiskFree* liefert -1 als Funktionsergebnis zurück, wenn *Drive* einen ungültigen Wert hat.

Querverweis *DiskSize*

Beispiel
```
uses WinDos;
```

```
begin
  Writeln('Auf dem momentan gesetzten Laufwerk sind ',
          DiskFree(0) div 1024, 'KByte verfügbar.');
end.
```

DiskSize-Funktion **WinDos**

Funktion	liefert die Gesamtkapazität eines Laufwerks zurück.
Deklaration	`DiskSize(Drive: Byte)`
Ergebnistyp	*LongInt*
Erläuterung	*Drive* ist ein Ausdruck des Typs *Byte* und gibt das Laufwerk an, dessen Kapazität (in Bytes) ermittelt werden soll. Der Wert 0 steht für das momentan gesetzte Laufwerk, der Wert 1 für Laufwerk A:, 2 für B: usw. *DiskSize* liefert -1 als Funktionsergebnis zurück, wenn *Drive* einen ungültigen Wert hat.
Querverweis	*DiskFree*

Beispiel	`uses WinDos;`
	`begin`
	`  Writeln('Das momentan gesetzte Laufwerk hat eine ',`
	`  'Kapazität von ', DiskSize(0) div 1024, 'KByte.');`
	`end.`

Dispose-Prozedur

Funktion	gibt den durch eine dynamische Variable belegten Speicherplatz auf dem Heap wieder frei.
Deklaration	`Dispose(var p: Pointer [ , Destrukt])`

Erläuterung	In der ersten Form der Deklaration ist *p* eine Zeigervariable beliebigen Typs, der zuvor mit der Prozedur *New* oder durch eine direkte Zuweisung ein Speicherbereich auf dem Heap zugeordnet wurde.
	Speziell für objektorientierte Programmierung wurde die Syntax von *Dispose* erweitert. In der zweiten Form der Deklaration ist *p* eine Zeigervariable auf einen Objekttyp und *Destrukt* ein Aufruf der Destruktor-Methode dieses Objekttyps. Diese erweiterte Syntax von *Dispose* ist nichts anderes als eine Zusammenfassung der beiden Anweisungen:
	`P^.Destrukt;`
	`Dispose(P);`
	Die erweiterte Syntax verbessert die Lesbarkeit und erzeugt außerdem einen kürzeren und effektiveren Code.
	Achtung! Der Wert von *p* ist nach dem Aufruf von *Dispose*

nicht mehr definiert, Zugriffe der Form *p*^ stellen einen Fehler dar.

Grenzen Wenn *p* nicht auf eine Adresse innerhalb des Heap-Bereichs zeigt, bricht das Programm mit einem Laufzeitfehler ab. *Dispose* und *FreeMem* können nur unter bestimmten Umständen zusammen mit *Mark* und *Release* verwendet werden (siehe Kapitel 16 »Speicherbelegung und Datenformate«).

Querverweis *FreeMem, GetMem, New*

Beispiel
```
type Str22 = string[22];
var  p : ^Str22;
begin
  New(p);
  p^:= 'Hier ist der Text...';
  Writeln(p^);
  Dispose(p);   { 'Weg ist der Text ... ' }
end.
```

DoneWinCrt-Prozedur **WinCrt**

Funktion entfernt ein CRT-Fenster, wenn dies nicht schon vorher entfernt wurde.

Deklaration `DoneWinCrt`

Erläuterung Wird *DoneWinCrt* unmittelbar vor dem Ende eines Programms aufgerufen, ist gewährleistet, daß das Fenster nicht deaktiviert wird. Der Benutzer muß dann das Fenster nicht selbst schließen.

DosVersion-Funktion **WinDos**

Funktion liefert die Versionsnummer von DOS zurück.

Deklaration `DosVersion`

Ergebnistyp *Word*

Erläuterung *DosVersion* liefert die Versionsnummer von DOS in Form eines *Word*-Wertes zurück, dessen Bytes voneinander getrennt interpretiert werden müssen: das höherwertige Byte enthält die Neben-Versionsnummer, das niederwertige Byte die Haupt-Versionsnummer. Für die DOS-Version 3.20 würde *DosVersion* also beispielsweise das Ergebnis $2003 zurückliefern.

Beispiel
```
program DosVersionDemo;
uses WinDos;
var
  Version: Word;
```

```
begin
  Version := DosVersion;
  Writeln
  ('Dieses Programm läuft unter der DOS-Version ',
           Lo(Version), '.', Hi(Version));
  end.
```

DSeg-Funktion

Funktion	liefert die Segmentadresse des Datensegments zurück.
Deklaration	`DSeg`
Ergebnistyp	*Word*
Erläuterung	*DSeg* liefert den Inhalt des Prozessor-Registers DS zurück, d.h. die Adresse des Segments, in dem die globalen Variablen des Programms gespeichert sind.
Querverweis	*CSeg*, *SSeg*

Eof-Funktion

Funktion	prüft, ob das Ende einer Datei erreicht ist.	
Deklaration	`Eof(var f)`	für typisierte und untypisierte Dateien
	Eof[(**var** `f: Text)`]	für Textdateien
Ergebnistyp	*Boolean*	

Erläuterung

Eof prüft, ob es in der angegebenen Datei ohne explizite Veränderung der momentanen Position »noch etwas zu lesen gibt«. In den folgenden Fällen wird eine Datei als »ausgelesen« betrachtet und *Eof* liefert das Ergebnis *True*:

- die Datei enthält überhaupt keine Komponenten (d.h. hat einen Umfang von 0 Bytes);
- die Datei ist nur für Schreibaktionen geöffnet worden oder es handelt sich um ein DOS-Gerät, das keine Leseaktionen zuläßt (wie beispielsweise der Bildschirm);
- der Positionszeiger innerhalb der Datei steht hinter der letzten Komponente.

Wenn *f* nicht angegeben ist, prüft Turbo Pascal die Standard-Eingabe: der Aufruf von *Eof* ohne Argument ist also identisch mit *Eof(Input)*.

Wenn die Prüfung durch *Eof* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im

Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Querverweis *Eoln*, *SeekEof*

Beispiel

```
program EofDemo;
{ Programm zur unformatierten Ausgabe von Textdateien }
var
  f : Text;
  ch : Char;
begin
  { Dateiname wird als 1. Kommandozeilen-
    Parameter vorausgesetzt }
  Assign(f, ParamStr(1));
  Reset(f);
  while not Eof(f) do  { solange das Dateiende
                         nicht erreicht ist }
  begin
   Read(f, ch); Write(ch);  { Ausgabe }
  end;
  Close(f);
end.
```

Eoln-Funktion

Funktion prüft, ob die Position innerhalb einer Datei auf einem Zeilenende steht.

Deklaration

`Eoln(var f)`	für typisierte und untypisierte Dateien
`Eoln[(var f: Text)]`	für Textdateien

Ergebnistyp *Boolean*

Erläuterung *Eoln* liefert *True* zurück, wenn der Positionszeiger innerhalb der durch *f* angegebenen Datei auf eine Zeilenende-Markierung zeigt oder das Dateiende erreicht ist (*Eof* = *True*).

Wenn *f* nicht angegeben ist, prüft Turbo Pascal die Standard-Eingabe: der Aufruf von *Eoln* ohne Argument ist also identisch mit *Eoln(Input)*.

Wenn die Prüfung durch *Eoln* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Querverweis *Eof*, *SeekEoln*

Erase-Prozedur

Funktion löscht eine Datei.

Deklaration `Erase(var f)`

Erläuterung *Erase* löscht die mit *f* (via *Assign*) verbundene externe Datei (wobei *f* für eine Datei-Variable beliebigen Typs steht).

Wenn *Erase* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Grenzen *Erase* darf nie auf offene Dateien angewendet werden.

Querverweis *Rename*

Beispiel
```
program EraseDemo;
  { Programm zum Löschen einer Datei OHNE Rückfrage }
var
  f : file;
begin
  { Name der zu löschenden Datei wird
    als 1.Kommandozeilen-Parameter vorausgesetzt }
  Assign(f, ParamStr(1));
  {$I-} Reset(f); {$I+}
  if IOResult <> 0 then
    Writeln('Nicht gefunden: ', ParamStr(1))
   else begin
    Close(f);   { Datei darf NICHT offen sein }
    Erase(f);
   end;
end.
```

Exit-Prozedur

Funktion verläßt den momentanen Block mit sofortiger Wirkung.

Deklaration `Exit`

Erläuterung die Ausführung von *Exit* innerhalb eines Unterprogramms (d.h. einer Prozedur oder Funktion) ist mit einem Rücksprung zur aufrufenden Routine identisch. Die Ausführung von *Exit* innerhalb des Hauptprogramms beendet das gesamte Programm. *Exit* hat im Prinzip denselben Effekt wie ein **goto** zu einem Label, das direkt vor dem jeweiligen Block-Ende steht.

Querverweis *Halt*

Beispiel
```
uses WinCrt;
procedure Warten;
begin
```

```
  repeat
   if KeyPressed then Exit;
   Write('Xx');
  until False;  { endlos }
end;
begin
  Warten;
end.
```

Exp-Funktion

Funktion	liefert *e* hoch dem Argument zurück.
Deklaration	`Exp(x: Real)`
Ergebnistyp	*Real*
Erläuterung	*x* ist ein Real-Ausdruck. Das Ergebnis der Funktion ist e^x, wobei *e* die Basis des natürlichen Logarithmus ist (2.718281...).
Querverweis	*Ln*

FileExpand-Funktion **WinDos**

Funktion	erweitert einen unvollständig angegebenen Dateinamen um den dazugehörigen Suchweg.
Deklaration	`FileExpand(Dest, Name: PChar)`
Ergebnistyp	*PChar*
Erläuterung	*FileExpand* erweitert den Dateinamen *Name* zu einem vollständigen Dateinamen. Der resultierende Name wird zu Großbuchstaben konvertiert und besteht aus Laufwerksbuchstabe, Verzeichnispfad und Dateiname. Verzeichnisverweise, die in '.'oder '..' eingeschlossen sind, werden entfernt. Name und Dateierweiterung werden auf acht und drei Zeichen verkürzt. Der gelieferte Wert ist *Dest. Dest* und *Name* können sich auf dieselbe Position beziehen. Wenn beispielsweise momentan das Verzeichnis C:\TURBO\DOC gesetzt ist, dann liefert *FileExpand* die folgenden Ergebnisse: FileExpand('test.pas') = 'C:\TURBO\DOC\TEST.PAS' FileExpand('..*.tpu') = 'C:\TURBO*.TPU' FileExpand('c:\bin*.txt') = 'C:\BIN*.TXT' *Hinweis:* Vollständige Dateinamen/Suchwege lassen sich mit der Prozedur *FileSplit* in ihre Komponenten »Suchweg«, »Dateiname« und »Suffix« zerlegen.
Querverweis	*FileSplit, FindFirst, FindNext*

FilePos-Funktion

Funktion	liefert die momentane Position innerhalb einer Datei zurück.
Deklaration	`FilePos(var f)`
Ergebnistyp	*LongInt*
Erläuterung	*f* steht für eine typisierte oder untypisierte Datei-Variable. Wenn die Position am Anfang der Datei steht (d.h. vor der ersten Komponente), dann liefert *FilePos* den Wert 0 zurück; steht sie auf dem Ende der Datei (*Eof* = *True*), dann ist *FilePos(f)* identisch mit *FileSize(f)*. Um die Byte-Position innerhalb der Datei zu erhalten, muß der von *FilePos* zurückgelieferte Wert mit der Recordgröße dieser Datei bzw. mit der Größe einer einzelnen Komponente multipliziert werden. Wenn *FilePos* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.
Grenzen	*FilePos* kann nicht auf Textdateien angewendet werden. Diese Funktion setzt voraus, daß die entsprechende Datei offen ist.
Querverweis	*FileSize*, *Seek*

FileSearch-Funktion **WinDos**

Funktion	sucht in den in einer Liste aufgeführten Verzeichnissen nach einer Datei.
Deklaration	`FileSearch (Dest, Name, Liste: PChar): PChar`
Ergebnistyp	*PChar*
Erläuterung	Sucht nach der Datei *Name* in der *Liste* mit den Unterverzeichnissen. Die Verzeichnisse in der *Liste* müssen genauso wie in der DOS-Path-Angabe durch Strichpunkte (;) voneinander getrennt sein. Die Suche beginnt immer im aktuellen Verzeichnis und im aktuellen Laufwerk. Wird die *Datei* gefunden, speichert *FileSearch* eine Verkettung des Verzeichnisnamens und des Dateinamens in *Dest*. Andernfalls speichert *FileSearch* einen leeren String in *Dest*. Der gelieferte Wert ist *Dest*. *Dest* und *Name* dürfen sich nicht auf denselben Speicherbereich beziehen. Die maximale Länge des Ergebnisses wird durch die Konstante *fsPathname* definiert. Sie hat den Wert 79. Um einen Pfad abzusuchen, der von DOS verwendet wird, um ausführbare Dateien zu speichern, muß *GetEnvVar*

('PATH') aufgerufen werden und das Ergebnis als der Parameter *Liste* an *FileSearch* übergeben werden.

Das Ergebnis von *FileSearch* kann an *FileExpand* übergeben werden, um es zu einem gültigen Dateinamen zu konvertieren, also einen großgeschriebenen Dateinamen, der sowohl den Buchstaben des Laufwerks als auch den Verzeichnispfad enthält. Außerdem kann *FileSplit* verwendet werden, um den Dateinamen in jeweils einen String für »Suchweg«, »Name« und »Suffix« aufzugliedern.

Querverweis *FileExpand, FileSplit*

Beispiel

```
uses WinCrt, WinDos
var
  S: array[0..fsPathname] of Char;
begin
  FileSearch (S, 'TPW.exe', GetEnvVar ('Path'));
  if S[0] = #0 then
    WriteLn('TPW.EXE nicht gefunden')
  else
   WriteLn('Gefunden als',FileExpand (S,S));
end.
```

FileSize-Funktion

Funktion liefert die Größe einer Datei zurück.

Deklaration `FileSize(var f)`

Ergebnistyp *LongInt*

Erläuterung *FileSize* liefert die Anzahl der Komponenten zurück, die in der angegebenen Datei gespeichert sind. *f* steht für eine typisierte oder untypisierte Datei-Variable. Wenn die Datei leer ist, liefert *FileSize* das Ergebnis 0.

Wenn *FileSize* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Grenzen *FileSize* kann nicht auf Textdateien angewendet werden. Die Funktion setzt voraus, daß die angegebene Datei offen ist.

Querverweis *FilePos*

Beispiel

```
program FileSizeDemo;
var
  f: file of Byte;
begin
  { Der Name der Datei wird als
    1. Kommandozeilen-Parameter erwartet }
```

```
  Assign(f, ParamStr(1));
  Reset(f);
  Writeln('Größe von ', ParamStr(1), ' in Bytes: ',
          FileSize(f));
  Close(f);
end.
```

FileSplit-Funktion **WinDos**

Funktion zerlegt einen vollständigen Dateinamen in seine drei Komponenten »Suchweg«, »Name« und »Suffix«.

Deklaration
```
FileSplit(Path, Dir, Name, Ext: PChar) : Word
```

Erläuterung *FileSplit* analysiert den als *Path* angegebenen String-Ausdruck und ordnet seine Komponenten den als *Dir*, *Name* und *Ext* übergebenen Stringvariablen zu. Wenn *Path* eine oder mehrere Komponenten nicht enthält, wird der entsprechenden Variablen ein Nullstring zugewiesen.
In *Dir* werden Laufwerksbezeichner und Suchweg (zusammen mit einem Doppelpunkt und den entsprechenden führenden bzw. folgenden umgekehrten Schrägstrichen) gespeichert; *Name* wird der Dateiname zugewiesen, *Ext* das Suffix mit einem führenden Punkt. Wenn ein Komponeten-Stringparameter **nil** ist, wird auch der entsprechende Teil des Pfads nicht gespeichert. Wenn der Pfad eine bestimmte Komponente nicht enthält, bleibt der gelieferte Komponentenstring leer. Die maximale Länge der von *Dir, Name* und *Ext* gelieferten Strings wird von den Konstanten *fsDirectory, fsFileName* und *fsExtension* bestimmt.

Der gelieferte Wert ist eine Kombination aus den Bitmasken *fsDirectory, fsFileName* und *fsExtension* und gibt an, welche Komponeten im Pfad vorhanden waren. Wenn der Name oder die Extension (Erweiterung) Jokerzeichen enthalten (* oder ?), wird das Flag *fcWildcard* im Rückgabewert der Funktion gesetzt.

Die Konstanten *fsXXXX* und *fcXXXX* werden in WinDos wie folgt definiert:

```
const
  fsPathName  = 79;
  fsDirectory = 67;
  fsFileName  = 8;
  fsExtension = 4;
```

```
const
 fsExtension = $0001;
 fsFileName  = $0002;
```

```
  fsDirectory = $0004;
  fsWildcards = $0008;
```

Querverweis *FileExpand, FindFirst, FindNext*

Beispiel

```
uses Strings, WinCrt, WinDos;
var
  Path: array [0..fsPathName] of Char;
  Dir: array [0..fsDirectory] of Char;
  Name: array [0..fsFileName] of Char;
  Ext: array [0..fsExtension] of Char;
begin
  Write('Filename (WORK.PAS): ');
  ReadLn(Path);
  FileSplit (Path, Dir, Name, Ext);
  if Name [0] = #0 then StrCopy (Name, 'WORK');
  if Ext [0] = #0 then StrCopy (Ext, '.PAS');
  StrECopy (StrECopy (StrECopy(Path, Dir), Name), Ext);
  WriteLn('Der Name lautet', Path);
end.
```

FillChar-Prozedur

Funktion füllt einen Speicherbereich mit einem bestimmten Wert.

Deklaration `FillChar(var x; count: Word; value)`

Erläuterung *x* steht für einen Variablenbezug beliebigen Typs, der von *FillChar* als Startadresse des zu füllenden Bereichs interpretiert wird. *count* gibt die Anzahl der mit *value* zu füllenden Speicherzellen an, *value* ist ein Ausdruck des Typs *Char* oder *Byte*.

Achtung! FillChar nimmt keine Bereichsprüfungen vor. Soweit möglich, sollte deshalb der Parameter *count* über *SizeOf* angegeben werden - auf diese Weise sorgt der Compiler dafür, daß *FillChar* nicht versehentlich andere Speicherbereiche mit überschreibt.

FillChar läßt sich nicht zuletzt zum Füllen von Strings verwenden. Bitte beachten Sie, daß das Längenbyte in diesem Fall explizit gesetzt werden muß (siehe »Beispiel«).

Querverweis *Move*

Beispiel

```
var
  s : string[80];
begin   { Setzen eines Strings mit 80 Sternchen }
  FillChar(s, SizeOf(s), '*');
  s[0] := #80;   { Längenbyte setzen }
end.
```

FindFirst-Prozedur WinDos

Funktion sucht ein Directory nach dem ersten Vorkommen eines Dateinamens ab.

Deklaration
```
FindFirst(Path: string; Attr: Byte; var S: SearchRec)
```

Erläuterung *Path* gibt den zu suchenden Dateinamen (z.B. ***.***) an und kann optional einen Laufwerksbezeichner und/oder einen Suchweg enthalten. Wenn *Path* nur einen Dateinamen angibt, benutzt *FindFirst* das momentan gesetzte Directory.

Attr legt die Attribute des zu suchenden Dateieintrags fest. Für diesen Parameter sind die folgenden Konstanten in der Unit *WinDos* deklariert:

```
const
  faReadOnly  = $01;
  faHidden    = $02;
  faSysFile   = $04;
  faVolumeID  = $08;
  faDirectory = $10;
  faArchive   = $20;
  faAnyFile   = $3F;
```

Bei der Angabe des Attributs müssen Sie beachten, daß *FindFirst* immer auch Dateien findet, die ein weniger einschränkendes Attribut haben. Wenn Sie z.B. *VolumeID* angeben, »findet« *FindFirst* das Plattenlabel, Subdirectories und normale Dateien, nicht aber verborgene und System-Dateien.

FindFirst erwartet einen (nicht initialisierten) Parameter des Typs *TSearchRec*, über dessen Felder die Ergebnisse der Suche zurückgeliefert werden. Der Typ *TSearchRec* ist ebenfalls im Unit *WinDos* definiert:

```
type
  TSearchRec = record
                 Fill: array[1..21] of Byte;
                 Attr: Byte;
                 Time: Longint;
                 Size: Longint;
                 Name: string[12];
               end;
```

Fehler bei der Ausführung von *FindFirst* setzen die globale Variable *DosError*. Mögliche Codes sind 0 (»fehlerfrei«), 2 (»Verzeichnis nicht gefunden«) und 18 (»Keine weiteren Einträge«).

Querverweis *FindNext*, *FileExpand*

Beispiel

```
program FindFirstDemo;
uses WinDos;
var
  DirInfo: TSearchRec;
begin
 FindFirst('*.PAS', Archive, DirInfo);
    { dasselbe wie DIR *.PAS }
 while DosError = 0 do
  begin
   Writeln(DirInfo.Name);
   FindNext(DirInfo);
  end;
end.
```

FindNext-Prozedur **WinDos**

Funktion setzt eine mit *FindFirst* begonnene Suche fort.

Deklaration `FindNext(var S: TSearchRec)`

Erläuterung *FindNext* erwartet eine Variable des Typs *TSearchRec*, die durch einen Aufruf von *FindFirst* initialisiert worden sein muß. Jeder Aufruf von *FindNext* sucht das jeweils nächste Vorkommen eines Dateieintrags, der die durch *FindFirst* festgelegten Bedingungen erfüllt, und liefert ihn in der übergebenen Variablen *s* zurück.

Fehler bei der Ausführung von *FindNext* setzen die globale Variable *DosError*: der einzig mögliche Code ist hier 18 (»keine weiteren Dateieinträge«).

Querverweis *FindFirst*, *FileExpand*

Beispiel siehe *FindFirst*

Flush-Prozedur

Funktion erzwingt das Schreiben des Dateipuffer-Inhalts einer Textdatei.

Deklaration `Flush(var f: Text)`

Erläuterung *f* steht für eine Textdatei-Variable. Normalerweise haben Ausgaben zu einer mit *Rewrite* oder *Append* geöffneten Textdatei erst dann eine physikalische Schreibaktion zur Folge, wenn der Dateipuffer voll ist. Ein Aufruf von *Flush* erzwingt das physikalische Schreiben des Pufferinhalts unabhängig von seinem »Füllungsgrad« (unter anderem wird diese Prozedur automatisch von *Close* aufgerufen).

Bei Dateien, die mit *Reset* eröffnet wurden (d.h. nur für Leseoperationen), hat *Flush* keine Wirkung.

Wenn *Flush* fehlerfrei ausgeführt werden konnte, dann

liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Grenzen — die durch *f* bezeichnete Datei muß offen sein.

Frac-Funktion

Funktion — liefert den nicht-ganzzahligen Teil des Arguments zurück.

Deklaration — `Frac(x: Real)`

Ergebnistyp — *Real*

Erläuterung — *x* ist ein Real-Ausdruck. Das Ergebnis der Funktion ist der nicht-ganzzahlige Anteil von *x* (d.h. die »Stellen nach dem Komma«) und läßt sich auch über die Formel *x - Int(x)* berechnen.

Querverweis — *Int*

Beispiel

```
var
  x : Real;
begin
  x := Frac(123.456);  { x := 0.456 }
end.
```

FreeMem-Prozedur

Funktion — gibt einen Speicherbereich bestimmter Größe auf dem Heap wieder frei.

Deklaration — `FreeMem(var p: Pointer; size: Word)`

Erläuterung — *p* ist eine Zeigervariable beliebigen Typs, der zuvor mit *New* oder durch eine direkte Zuweisung ein Speicherbereich auf dem Heap zugeordnet wurde. *Size* ist ein Integer-Ausdruck, gibt die Größe dieses Speicherbereichs (bzw. der Variablen) in Bytes an und muß **exakt** mit der Größe des über *p* belegten Speicherbereichs übereinstimmen.

Der Wert von *p* ist nach dem Aufruf von *Dispose* nicht mehr definiert, Zugriffe der Form *p^* stellen einen Fehler dar.

Querverweis — *Dispose*, *GetMem*

GetArgCount-Funktion WinDos

Funktion liefert die Anzahl der Parameter zurück, die dem Programm in der Kommandozeile übergeben wurden.

Deklaration
```
GetArgCount: Integer
```

GetArgStr-Funktion WinDos

Funktion liefert den Kommandozeilen-Parameter zurück, der mit *Index* angegeben wurden.

Deklaration
```
GetArgStr(Dest: PChar; Index: Integer; MaxLen: Word):
PChar;
```

Erläuterung Wenn *Index* kleiner als Null oder größer als *GetArgCount* ist, liefert *GetArgStr* einen leeren String zurück. Wenn *Index* gleich Null ist, liefert *GetArgStr* den Dateinamen des aktuellen Moduls. *Dest* ist der gelieferte Wert. Die maximale Länge des gelieferten Strings wird mit dem Parameter *MaxLen* angegeben.

Querverweis *GetArgCount*

GetCBreak-Prozedur WinDos

Funktion ermittelt, bei welchen Operationen DOS auf `[Ctrl][Break]` prüft.

Deklaration
```
GetCBreak(var Break: Boolean)
```

Erläuterung *GetCBreak* ermittelt, ob DOS nur bei Ein-/Ausgaben oder bei jedem Aufruf auf die Eingabe von `[Ctrl][Break]` prüft. Wenn ausgeschaltet *(False)*, wird nur während Ein-/Ausgabeoperationen auf Konsolen-, Drucker- oder Kommunikationsgeräten geprüft. Wenn eingeschaltet *(True)* wird jeder Systemaufruf geprüft.

Querverweis *SetCBreak*

GetCurDir-Funktion WinDos

Funktion liefert das aktuelle Verzeichnis im aktuellen Laufwerk.

Deklaration
```
GetCurDir(Dir: PChar; Drive: Byte): PChar
```

Ergebnistyp *PChar*

Erläuterung Der in *Dir* gelieferte Wert beginnt immer mit dem Laufwerksnamen, einem Doppelpunkt und einem Backslash (\). Drive = 0 gibt das aktuelle Laufwerk an, 1 gibt Laufwerk A an, 2 gibt Laufwerk B an etc. Der gelieferte Werte ist *Dir*. Fehler werden in *DosError* angegegeben.

Die maximale Anzahl der Ergebnisstrings wird mit der Konstanten *fsDirectory* definiert.

Querverweis *CreateDir, RemoveDir, SetCurDir. GetDir* liefert das aktuelle Verzeichnis eines bestimmten Laufwerks als Pascal-String.

GetDate-Prozedur **WinDos**

Funktion ermittelt das momentan gesetzte Kalenderdatum.

Deklaration
```
GetDate(var Year, Month, Day, DayofWeek: Word)
```

Erläuterung *GetDate* liefert die Werte für das Kalenderdatum zurück, die von DOS benutzt werden. *Year* hat einen möglichen Wertebereich von 1980..2099, *Month* von 1..12, *Day* von 1..31 und *DayofWeek* (Wochentag) von 0..6, wobei der Wert 0 für Sonntag steht.

Querverweis *GetTime*, *SetDate*, *SetTime*

GetDir-Prozedur

Funktion ermittelt das momentan gesetzte Verzeichnis eines Laufwerks.

Deklaration
```
GetDir(d: Byte; var s: string)
```

Erläuterung *d* ist ein Ausdruck des Typs *Integer* und gibt das Laufwerk an (0 = momentan gesetztes Laufwerk, 1 = Laufwerk A:, 2 = B: usw); *s* ist eine String-Variable und bekommt von *GetDir* den Namen bzw. Suchweg des Verzeichnisses zugewiesen, das für das entsprechende Laufwerk aktuell gesetzt ist.

GetDir an sich führt keine Fehlerprüfung durch. Wenn das mit *d* angegebene Laufwerk ungültig ist, liefert *s* '\' zurück, als ob es das Stammverzeichnis des ungültigen Laufwerks sei.

Querverweis *ChDir, DiskFree, DiskSize, MkDir, RmDir. GetCurDir* hat dieselbe Funktion wie *GetDir*, benötigt aber einen null-terminierten String als Argument und keinen Pascal-String.

GetEnvVar-Funktion **WinDos**

Funktion liefert einen Zeiger auf den Wert einer angegebenen Environment-Variable.

Deklaration
```
GetEnvVar(VarName: PChar)
```

Ergebnistyp *PChar*

Erläuterung *GetEnvVar* liefert einen Zeiger auf den Wert der angegebenen Variable (zum Beispiel einen Zeiger auf das erste Zeichen nach einem Gleichheitszeichen (=) im von *VarName* definierten

Umgebungseintrag. Der Variablenname kann entweder groß- oder kleingeschrieben werden, darf aber kein Gleichheitszeichen enthalten. Wenn die angegebene Umgebungsvariable nicht existiert, liefert *GetEnvVar* **nil**.

Beispiel

```
uses WinCrt, WinDos;
begin
  WriteLn('Der aktuelle Pfad ist ', GetEnvVar ('Path'));
end.
```

GetFAttr-Prozedur — WinDos

Funktion — liefert die Attribute einer Datei zurück.

Deklaration

```
GetFAttr(var f; var Attr: Word)
```

Erläuterung — *f* steht für eine Datei-Variable beliebigen Typs, der mit *Assign* eine externe Diskettendatei zugeordnet worden ist. *GetFAttr* ermittelt die Attribute dieser Datei, die *nicht* offen sein darf, und setzt die übergebene Variable *Attr* entsprechend.

Die einzelnen Attribute lassen sich durch **and**-Operationen mit den folgenden (im Unit *WinDos* definierten) Konstanten ermitteln:

```
const
  faReadOnly  = $01;
  faHidden    = $02;
  faSysFile   = $04;
  faVolumeID  = $08;
  faDirectory = $10;
  faArchive   = $20;
  faAnyFile   = $3F;
```

Fehler während der Ausführung von *GetFAttr* setzen die globale Variable *DosError*, mögliche Codes sind 3 (»ungültiger Suchweg«) und 5 (»Dateizugriff verweigert«).

Grenzen — die durch *f* bezeichnete Datei darf nicht offen sein.

Querverweis — *SetFAttr*, *GetFTime*, *SetFTime*

Beispiel

```
program GetFAttrDemo;
uses WinDos;
var
  f : file;
  attr: Word;
begin
  { Dateiname wird als
    1. Kommandozeilen-Parameter erwartet }
  Assign(f, ParamStr(1));
  GetFAttr(f, attr);
```

```
  Writeln('Datei: ', ParamStr(1));
  if DosError <> 0 then
    Writeln ('DOS-Fehler: ', DosError)
   else begin
    Write ('Attribute: ');
   if attr and ReadOnly <> 0 then
       Write ('schreibgeschützt  ');
   if attr and Hidden <> 0 then Write ('"hidden"  ');
   if attr and SysFile <> 0 then Write ('System  ');
   if attr and VolumeID <> 0 then Write ('Vol-ID  ');
   if attr and Directory <> 0 then Write ('(Sub)Dir  ');
   if attr and Archive <> 0 then
       Write ('Archive (normale Datei)');
    Writeln;
   end; { else }
end.
```

GetFTime-Prozedur **WinDos**

Funktion ermittelt Datum und Uhrzeit der letzten Veränderung einer Datei.

Deklaration `GetFTime(var f; var Time: LongInt)`

Erläuterung *f* ist eine Datei-Variable beliebigen Typs, die für eine externe und offene Diskettendatei steht. Der über die Variable *Time* zurückgelieferte Wert enthält Datum und Uhrzeit in einem »gepackten« Format und kann mit der Prozedur *UnpackTime* interpretiert werden.

Fehler bei der Ausführung von *GetFTime* setzen die globale Variable *DosError*; der einzig mögliche Code ist 6 (»ungültiger Handle/Datei nicht offen«).

Grenzen die durch *f* bezeichnete Datei muß offen sein.

Querverweis *SetFTime*, *SetAttr*, *PackTime*, *UnpackTime*

GetIntVec-Prozedur **WinDos**

Funktion ermittelt den Inhalt eines Interrupt-Vektors.

Deklaration `GetIntVec(IntNo: Byte; var Vector: Pointer)`

Erläuterung *IntNo* ist ein Ausdruck des Typs *Byte* und gibt die Nummer des zu ermittelnden Interrupt-Vektors (0..255) an, dessen Inhalt von *GetIntVec* in der als *Vector* übergebenen Zeigervariablen gespeichert wird.

Querverweis *SetIntVec*

GetMem-Prozedur

Funktion	belegt einen Bereich bestimmter Größe auf dem Heap.
Deklaration	`GetMem(var p: Pointer; Size: Word)`
Erläuterung	*GetMem* belegt einen Bereich von *Size* Bytes auf dem Heap und weist der als *p* übergebenen Zeigervariablen die Startadresse dieses Bereichs zu, erzeugt also eine dynamische Variable, auf die über *p^* zugegriffen werden kann. *p* ist eine Zeigervariable beliebigen Typs, *Size* ist ein Ausdruck des Typs *Word*. Wenn der Heap nicht mehr genügend Platz für die Belegung von *Size* Bytes (in einem zusammenhängenden Block) bietet, bricht die Programmausführung mit einem Laufzeitfehler ab. (siehe auch Kapitel 16 »Speicherbelegung und Datenformate«.
Grenzen	maximal können mit einem Aufruf von *GetMem* 65521 Bytes (= 64 KByte-$0F) belegt werden.
Querverweis	*FreeMem*, *Dispose*, *New*

GetTime-Prozedur **WinDos**

Funktion	liefert die Uhrzeit des Systems zurück.
Deklaration	`GetTime(var Hour, Min, Second, Sec100: Word)`
Erläuterung	GetTime setzt die übergebenen Variablen mit den Daten des Betriebssystems. Mögliche Werte für *Hour* sind 0..23, für *Minute* und *Second* 0..59, für *Sec100* (hundertstel Sekunden) 0..99.
Querverweis	*GetDate*, *SetDate*, *GetTime*, *UnpackTime*

GetVerify-Prozedur **WinDos**

Funktion	ermittelt den Stand des DOS-Flags *Verify*.
Deklaration	`GetVerify(var Verify: Boolean)`
Erläuterung	über das Flag *Verify* wird festgelegt, ob DOS geschriebene Diskettensektoren automatisch überprüft oder nicht. (Normalerweise ist dieses Flag auf *False* gesetzt eine direkte Überprüfung findet also nicht statt). *GetVerify* kopiert das Flag von *Dos* in die als *Verify* übergebene Variable: *True* bedeutet, daß DOS alle Diskettensektoren nach dem Schreiben noch einmal liest und auf diese Weise Prüfsummen verifiziert.
Querverweis	*SetVerify*

GotoXY-Prozedur **WinCrt**

Funktion setzt den Cursor auf die angegebene Position im virtuellen Bildschirm.

Deklaration `GotoXY(X, Y: Byte)`

Erläuterung Die obere linke Ecke entspricht den Koordinaten (1,1). Die Variable *Cursor* wird auf die Position (X-1, Y-1) gesetzt, da sie die Position relativ zu (0,0) und nicht relativ zu (1,1) speichert.

Halt-Prozedur

Funktion bricht die Ausführung eines Programms ab und kehrt zur Betriebssystemebene zurück.

Deklaration `Halt [(Exitcode: Word)]`

Erläuterung *Exitcode* ist ein (optionaler) Ausdruck des Typs *Word* und kann benutzt werden, um den Exit-Code einen Programms anzugeben.

Hinweis: Beachten Sie, daß *Halt* die Ausführung eventuell definierter Exit-Prozeduren initiiert (siehe Kapitel 18 »Interne Details).

Querverweis *Exit*, *RunError*

Hi-Funktion

Funktion liefert das höherwertige Byte des Arguments zurück.

Deklaration `Hi(x)`

Ergebnistyp *Byte*

Erläuterung *x* ist ein Ausdruck mit dem Typ *Integer* oder *Word*. Das Funktionsergebnis ist das höherwertige Byte des Wertes von *x* und ist vorzeichenlos.

Querverweis *Lo*, *Swap*

Beispiel
```
var  w: Word;
begin
  w := Hi($1234);   { w := $12; }
end.
```

Inc-Prozedur

Funktion erhöht eine Variable.

Deklaration `Inc(x [, n: LongInt])`

Erläuterung *x* ist eine Variable ordinalen Typs, der optionale Parameter *n* ist ein Integer-Ausdruck. Wenn *n* nicht angegeben ist, erhöht *Inc* die Variable *x* um 1 (und entspricht damit der Zuweisung `x := x+1`), ansonsten wird *x* um *n* erhöht (`x := x+n`).

Inc erzeugt optimierten Code. Die Vorteile dieser »Abkürzung« kommen am meisten in engen Schleifen und bei komplexen Adressierungen zum Tragen.

Querverweis *Dec*, *Pred*, *Succ*

Beispiel
```
var
  IntVar : Integer;
  LongVar: LongInt;
begin
  IntVar := 10; LongVar := 23948723;
  Inc(IntVar);           { IntVar := IntVar + 1 }
  Inc(LongVar,5);        { LongVar := LongVar + 5);
end.
```

InitWinCrt-Funktion **WinCrt**

Funktion erzeugt ein CRT-Fenster, falls dies noch nicht existiert.

Deklaration `WinInitCrt`

Erläuterung *Read, ReadLn, Write* oder *WriteLn* mit einer Datei, die einem CRT-Fenster zugewiesen ist, ruft automatisch *InitWinCrt* auf, um sicherzustellen, daß das CRT-Fenster auch existiert. *InitWinCrt* verwendet die Variablen *WindowOrg, WindowSize, ScreenSize* und *WindowTitle*, um die Charakteristika des CRT-Fensters festzulegen.

Insert-Prozedur

Funktion fügt einen Stringteil in einen String ein.

Deklaration `Insert(source: string; var s: string; index: Integer)`

Erläuterung *source* ist ein String-Ausdruck, *index* ein Integer-Ausdruck. *s* ist eine String-Variable, in die *Insert* den Inhalt von *source* ab der durch *index* angegebenen Zeichenposition einfügt. Wenn index größer als die Länge von *s* ist, hat *Insert* denselben Effekt wie *Concat*.

Insert begrenzt die Länge von *s* automatisch auf 255, überzählige Zeichen werden (ohne Fehlermeldung) abgeschnitten.

Querverweis *Delete*, *Copy*, *Concat*, *Length*, *Pos*

Beispiel
```
var
  s : string
begin
```

```
  s := 'Das ist artig!';
  Insert('groß', s, 9);  { s := 'Das ist großartig!' }
end.
```

Int-Funktion

Funktion	liefert den ganzzahligen Anteil des Arguments zurück.
Deklaration	`Int(x: Real)`
Ergebnistyp	*Real*
Erläuterung	*x* ist ein Ausdruck des Typs *Real*. Das Funktionsergebnis ist der ganzzahlige Anteil von *x*, gerundet gegen Null.
Querverweis	*Frac*, *Trunc*, *Round*
Beispiel	

```
var
  r : Real;
begin
  r := Int(123.456);   { r := 123.0 }
  r := Int(-123.456);  { r := -123.0 }
end.
```

Intr-Prozedur **WinDos**

Funktion	führt einen Software-Interrupt aus.
Deklaration	`Intr(IntNo: Byte; var Regs: TRegisters)`
Erläuterung	*IntNo* ist ein Ausdruck des Typs *Byte* und gibt die Nummer des Interrupts (0..255) an, der ausgeführt werden soll. Mit der als *Regs* übergebenen Variablen werden die Register des Prozessors vor der Ausführung des Interrupt-Befehls gesetzt. Nach der Ausführung speichert *Intr* die Prozessor-Register wieder in *Regs* zurück. (Diese Variable dient sowohl zur Übergabe von Parametern als auch zum Speichern des Ergebnisses). Der Datentyp *TRegisters* ist in der Unit *WinDos* folgendermaßen definiert:

```
type
  TRegisters = record
    case Integer of
      0 : (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Word);
      1 : (AL,AH,BL,BH,CL,CH,DL,DH: Byte);
    end;
```

Hinweis: Um Protection-Fehler im Standard- und erweiterten 386er-Modus von Windows zu vermeiden, sollten Sie immer sicherstellen, daß die Felder *DS* und *ES* des Records *TRegister* mit gültigen Selector-Werten oder mit Null initialisiert sind.

Grenzen Software-Interrupts, die bestimmte Werte der Register SS:SP voraussetzen (oder diese Register verändern), können über *Intr* nicht ausgeführt werden.

Querverweis *MsDos*

IOResult-Funktion

Funktion liefert den Fehlerstatus der letzten Ein-/Ausgabeoperation zurück.

Deklaration `IOResult`

Ergebnistyp *Word*

Erläuterung Im Modus {$I-} unterbleibt diese automatische Prüfung: hier kann (und muß) das Programm mögliche Fehler über den Aufruf von *IOResult* selbst prüfen. Mögliche Ergebnisse (Fehlercodes) sind in Anhang A zusammengefaßt der Wert 0 steht für »fehlerfreie Ausführung«.

Hinweis: Nach einem Fehler werden folgende Ein- und Ausgabebefehle solange ignoriert, bis das Programm *IOResult* aufgerufen hat.

Jeder Aufruf von *IOResult* setzt die intern gespeicherte Fehlernummer auf 0 zurück.

Beispiel

```
program IOResultDemo;
var
  f : file of Byte;
begin
 { Dateiname wird als
   1. Kommandozeilen-Parameter vorausgesetzt }
  Assign(f, ParamStr(1));
  {$I-} Reset(f); {$I+}
  if (IOResult = 0) then
    Writeln('Dateigröße: ', FileSize(f), ' Bytes.')
   else
    Writeln('Datei nicht gefunden.');
end.
```

KeyPressed-Funktion **WinCrt**

Funktion prüft, ob eine Taste gedrückt wurde.

Deklaration `KeyPressed: Boolean`

Ergebnistyp *Boolean*

Erläuterung *KeyPressed* liefert *True*, wenn die Taste gedrückt wurde, und *False*, wenn keine Taste gedrückt wurde. Die Taste kann mit der Funktion *ReadKey* gelesen werden.

Querverweis *ReadKey*

Length-Funktion

Funktion liefert die dynamische Länge eines Strings zurück.

Deklaration `Length(s: string)`

Ergebnistyp *Integer*

Erläuterung *s* ist ein String-Ausdruck. Das Funktionsergebnis entspricht der Anzahl der in *s* enthaltenen Zeichen.

Querverweis *Concat, Copy, Delete, Insert, Pos*

Beispiel

```
var
  s : string;
  i : Integer;
begin
  s := 'abc';
  for i := 1 to Length(s) do
   s[i] := UpCase(s[i]);    { ergibt 'ABC' }
  Writeln(s);
end.
```

Ln-Funktion

Funktion liefert den natürlichen Logarithmus des Arguments zurück.

Deklaration `Ln(x: Real)`

Ergebnistyp *Real*

Erläuterung *x* ist ein Real-Ausdruck. Das Funktionsergebnis ist der natürliche Logarithmus von *x*.

Querverweis *Exp*

Lo-Funktion

Funktion liefert das niederwertige Byte des Arguments zurück.

Deklaration `Lo(x)`

Ergebnistyp *Byte*

Erläuterung *x* ist ein Ausdruck des Typs *Integer* oder *Word*. Das Funktionsergebnis ist das niederwertige Byte des Wertes von *x* und ist vorzeichenlos.

Querverweis *Hi, Swap*

Beispiel

```
var
  w : Word;
begin
  w := Lo($1234);   { w := $34 }
end.
```

MaxAvail-Funktion

Funktion liefert den Umfang des größten freien Blocks im Heap zurück.

Deklaration `MaxAvail`

Ergebnistyp *LongInt*

Erläuterung *MaxAvail* vergleicht die Größe des größten freien Blocks im zur Unterverteiling verfügbaren Platz des Heapmanagers mit dem globalen Heap von Windows und liefert den größeren der beiden Werte zurück. Dieser entspricht der größten dynamischen Variablen, der zu diesem Zeitpunkt mit *New* oder *GetMem* Speicherplatz zugewiesen werden könnte. Die Größe des größten freien Blocks auf dem globalen Heap von Windows wird mit der Windows-Funktion *GlobalCompact* berechnet.

Querverweis *MemAvail*

Beispiel

```
program MaxAvailDemo;
type
  PBuffer = ^TBuffer;
  TBuffer = array[0..16383] of Char;
var
  Buffer: PBuffer;
begin
  .
  .
  if MaxAvail  SizeOf(TBuffer) then OutOfMemory else
   begin
     New(Buffer);
     .
     .
   end;
   .
   .
end.
```

MemAvail-Funktion

Funktion	liefert die Gesamtmenge des noch freien Speicherplatzes auf dem Heap zurück.
Deklaration	`MemAvail`
Ergebnistyp	*LongInt*
Erläuterung	*MemAvail* berechnet den freien Speicherplatz durch Aufruf der Windows-Funktion *GetFreeSpace* und addiert die Größen sämtlicher freien Blöcke im zur Unterverteilung zur Verfügung stehenden Bereich des Heap-Managers.
Querverweis	*MaxAvail*

Beispiel	`begin`
	`  Writeln ('Insgesamt ', MemAvail, 'Bytes verfügbar');`
	`  Writeln ('Größter Block: ', MaxAvail, ' Bytes');`
	`end.`

MkDir-Prozedur

Funktion	erzeugt ein Unterverzeichnis.
Deklaration	`MkDir(s: string)`
Erläuterung	*s* ist ein String-Ausdruck und gibt den Namen des neu zu erzeugenden Unterverzeichnisses an, optional auch einen Laufwerksbezeichner und einen Suchpfad. Das letzte Element des Pfads darf kein bereits existierender Dateiname sein. Wenn *MkDir* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.
Querverweis	*RmDir*, *ChDir*, *GetDir*. *CreateDir* hat dieselbe Funktion wie *MkDir*, benötigt aber einen null-teminierten String als Argument und keinen Pascal-String.

Beispiel	`program MkDirDemo;`
	`begin`
	`  { Directory-Name wird als`
	`    1. Kommandozeilen-Parameter angenommen }`
	`  {$I-} MkDir(ParamStr(1)); {I+}`
	`  if IOResult <> 0 then`
	`    Writeln('Verzeichnis konnte nicht erzeugt werden!')`
	`   else`
	`    Writeln('Verzeichnis ', ParamStr(1), 'erzeugt.');`
	`end.`

Move-Prozedur

Funktion kopiert Bytes von einem Speicherbereich in einen anderen.

Deklaration
```
Move(var Source, Dest; count: Word)
```

Erläuterung *Source* und *Dest* sind Variablenbezüge beliebigen Typs und werden von *Move* als Startadressen des Quellbereichs (*source*) bzw. Zielbereichs (*dest*) interpretiert. *count* ist ein Ausdruck des Typs *Word* und gibt an, wieviele Bytes von *Source* nach *Dest* kopiert werden sollen.

Achtung! Move nimmt keine Prüfung des Zielbereichs vor. Wo immer möglich, sollte deshalb die Anzahl der zu kopierenden Bytes über *SizeOf* bestimmt werden.

Hinweis: Wenn sich Ziel- und Quellbereich innerhalb desselben Segments befinden, sorgt *Move* automatisch für die richtige Richtung des Kopiervorgangs (Kopie mit aufsteigenden/absteigenden Adressen). Da dynamischen Variablen durch die Normalisierung immer eine eigene Segmentadresse zugewiesen wird, kann es auf dem Heap (außer nach direkten Eingriffen des Programmierers) keine Überlappungen geben folglich arbeitet *Move* bei unterschiedlichen Segmentadressen ohne weitere Prüfungen.

Querverweis *FillChar*

Beispiel
```
var
  a : array[1..4] of Byte;
  b : LongInt;
begin
  Move(a, b, SizeOf(a));
end.
```

MsDos-Prozedur **WinDos**

Funktion führt einen Funktionsaufruf von DOS aus.

Deklaration
```
MsDos(var Regs: TRegisters)
```

Erläuterung der Aufruf der Prozedur *MsDos* hat dieselbe Wirkung wie ein Aufruf der Prozedur *Intr* mit *IntNo = $21*. Vor der Ausführung des Interrupts werden die Register des Prozessors mit den als *Regs* übergebenen Werten gesetzt, nach Ausführung des Interrupts schreibt *MsDos* die Register-Inhalte in die Felder dieses **var**-Parameters zurück. Der Datentyp *TRegisters* ist im Unit *WinDos* folgendermaßen definiert:

```
type
  TRegisters = record
     case Integer of
       0 : (AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Word);
```

```
    1 : (AL,AZH,BL,BH,CL,CH,DL,DH: Byte);
end;
```

Grenzen Software-Interrupts, die bestimmte Werte der Register SS:SP voraussetzen oder diese Register verändern, können über *MsDos* nicht ausgeführt werden.

Querverweis *Intr*

New-Prozedur

Funktion erzeugt eine dynamische Variable und setzt einen Zeiger darauf.

Deklaration `New(var P: Pointer) [ , Init: Konstrukt ] )`

Erläuterung In der ersten Form der Deklaration ist *P* eine Zeigervariable beliebigen Typs. *New* belegt einen Speicherbereich auf dem Heap, dessen Größe durch die Typen-Deklaration von *P* vorgegeben ist, und weist *P* die Startadresse dieses Bereichs zu, auf den dann über P^ zugegriffen werden kann. Steht auf dem Heap nicht mehr genügend Speicher für die neue Variable zur Verfügung, wird ein Laufzeitfehler ausgelöst. (Es ist in diesem Fall möglich, einen Laufzeitfehler zu vermeiden. Vergleichen Sie dazu in Kapitel 16, »Speicherbelegung und Datenformate«, den Abschnitt »Die Variable HeapError«.)

Speziell für die objektorientierte Programmierung wurde die Syntax von *New* so erweitert, daß als zweiter Parameter ein Konstruktor zur Instantiierung einer dynamischen Objekt-Variablen übergeben werden kann. In dieser Form der Deklaration ist *P* eine Zeigervariable auf einen Objekttyp und *Konstrukt* ein Aufruf der Konstruktor-Methode dieses Objekttyps.

Eine weitere Erweiterung erlaubt die Verwendung von *New* als *Funktion*, die eine dynamische Variable eines bestimmten Typs – darin eingeschlossen Objekttypen – anlegt und einen Zeiger darauf zurückgibt. Die Syntax lautet dann: *New(P)P* oder *New(P, Init)*.

In der ersten Form kann *P* ein beliebiger Zeigertyp sein. In der zweiten Form muß *P* auf einen Objekttyp zeigen. *Init* ist der Aufruf des Konstruktors dieses Objekttyps. In beiden Fällen ist der Typ des Funktionsergebnisses *P*.

Querverweis *GetMem*, *Dispose*, *FreeMem*

Odd-Funktion

Funktion	prüft, ob das übergebene Argument eine ungerade Zahl darstellt.
Deklaration	`Odd(x: LongInt)`
Ergebnistyp	*Boolean*
Erläuterung	*x* ist ein Ausdruck des Typs *LongInt*. *Odd* liefert den Wert *True* zurück, wenn *x* ungeradzahlig ist; ansonsten ist das Funktionsergebnis *False*.

Ofs-Funktion

Funktion	liefert den Offset-Anteil einer Adresse zurück.
Deklaration	`Ofs(x)`
Ergebnistyp	*Word*
Erläuterung	*x* steht für einen Variablenbezug beliebiger Art oder den Bezeichner einer Prozedur oder Funktion. Das Ergebnis von *Ofs* ist der *Offset*-Anteil der durch *x* angegebenen Adresse.
Querverweis	*Seg*, *Addr*

Ord-Funktion

Funktion	liefert die Ordinalzahl des Arguments zurück.
Deklaration	`Ord(x)`
Ergebnistyp	*LongInt*
Erläuterung	*x* ist ein Ausdruck ordinalen Typs. Das Ergebnis der Funktion ist die Ordinalzahl von *x*.
Querverweis	*Chr*
Beispiel	

```
var
  Wurst: (Kohl, Birne);
begin
  Writeln(Ord(Kohl));       { 0 }
  Writeln(Ord('A'));      { 65 }
end.
```

PackTime-Prozedur **WinDos**

Funktion konvertiert einen Record des Typs *TDateTime* in einen »gepackten« *LongInt* für die Verwendung in *SetFTime*.

Deklaration `PackTime(var DT: TDateTime; var Time: LongInt)`

Erläuterung der Datentyp *TDateTime* ist in der Unit *WinDos* folgendermaßen deklariert;

```
TDateTime = record
    Year, Month,   { Jahr 1980..2099, Monat 1..12 }
    Day, Hour,     { Tag 1..31, Stunde 0..23 }
    Min, Sec: Word;  { Minuten, Sekunden 0..59 }
  end;
```

Querverweis *UnpackTime, GetFTime, GetTime, SetFTime, SetTime*

ParamCount-Funktion

Funktion liefert die Anzahl der Kommandozeilen-Parameter zurück, die dem Programm übergeben wurden.

Deklaration `ParamCount`

Ergebnistyp *Word*

Erläuterung Leerzeichen und Tabs werden als Trennung zwischen zwei Parametern betrachtet.

Querverweis *ParamStr*

Beispiel

```
var
  x : Word;

begin
  if ParamCount = 0 then
    Writeln('Heute keine Parameter.')
  else
   for x:= 1 to ParamCount do
    Writeln('Parameter ',x,': ', ParamStr(x));
end.
```

ParamStr-Funktion

Funktion liefert einen Kommandozeilen-Parameter zurück.

Deklaration `ParamStr(index)`

Ergebnistyp *String*

Erläuterung *index* ist ein Ausdruck des Typs *Word* und gibt die Nummer des

Kommandozeilen-Parameters an, der von *ParamStr* zurückgeliefert werden soll.

Wenn *index* größer als *ParamCount* ist, dann liefert *ParamStr* einen Nullstring als Ergebnis.

Hinweis: Beim Betrieb unter DOS 3.x liefert *ParamStr(0)* Suchweg und Namen des laufenden Programms. Unter DOS 2.x hat *ParamStr(0)* einen Nullstring als Ergebnis.

Querverweis *ParamCount*

Beispiel
```
var I: Word;
begin
  for I := 1 to ParamCount do
    Writeln(ParamStr (I));
end.
```

Pi-Funktion

Funktion liefert den Wert der mathematischen Konstanten *Pi* zurück (3.141592653589793285).

Deklaration `Pi`

Ergebnistyp *Real*

Erläuterung das Ergebnis der Funktion ist der Wert von *Pi*.

Hinweis: Die Genauigkeit hängt davon ab, ob der Compiler im 8087-Modus oder im reinen Software-Modus arbeitet.

Pos-Funktion

Funktion sucht einen String nach einer Zeichenfolge ab.

Deklaration `Pos(substr : string; s: string)`

Ergebnistyp *Byte*

Erläuterung *Pos* sucht den als *s* übergebenen String-Ausdruck nach dem ersten Vorkommen des String-Ausdrucks *substr* ab. Wenn *substr* in *s* vollständig enthalten ist, dann liefert *Pos* die Position des Zeichens in *s* zurück, mit dem *substr* beginnt. Wenn *substr* nicht in *s* enthalten ist, dann hat *Pos* das Funktionsergebnis 0.

Querverweis *Concat, Copy, Delete, Insert, Length*

Beispiel
```
var
  s : string;
begin
  s := 'ABCDEF';
  Writeln(Pos('BCD', s));   { Ausgabe: 2 }
  Writeln(Pos('ACD', s));   { Ausgabe: 0 }
```

```
  Writeln(Pos('EFG', s));   { Ausgabe: 0 }
end.
```

Pred-Funktion

Funktion	liefert den Vorgänger des Arguments zurück.
Deklaration	`Pred(x)`
Ergebnistyp	derselbe Typ wie das Argument.
Erläuterung	*x* ist ein Ausdruck ordinalen Typs. Das Funktionsergebnis ist der Vorgänger von *x* (und hat denselben Typ).
Grenzen	wenn *x* das Element mit dem niedrigsten ordinalen Wert innerhalb seines Typs darstellt, dann ist der Vorgänger von *x* nicht definiert, und *Pred* erzeugt einen Laufzeitfehler.
Querverweis	*Dec*, *Inc*, *Succ*

Ptr-Funktion

Funktion	konvertiert zwei Angaben für *Segment* und *Offset* in einen Wert des Typs *Pointer*.
Deklaration	`Ptr(Seg, Ofs: Word)`
Ergebnistyp	*Pointer*

Erläuterung	*Seg* und *Ofs* sind Ausdrücke des Typs *Word*. Das Funktionsergebnis ist ein Zeiger, der auf die durch *Seg* und *Ofs* angegebene Adresse zeigt (*Segment:Offset*). Wie die Konstante **nil** ist das Ergebnis von *Ptr* ein »generischer Zeiger« und mit allen anderen Zeiger-Typen kompatibel.
	Das Funktionsergebnis kann zusammen dereferenziert und typisiert werden:
	`if Byte(Ptr($40, $49)^ ) ) = 7 then`
	`  Writeln('Der momentan gesetzte Videomodus = mono ');`

Querverweis	*Addr, Ofs, Seg*

Beispiel	`var`
	`  p : ^Byte;  { Zeiger auf Variablen des Typs Byte }`
	`begin`
	`  p := Ptr($40,$49);  { implizit konvertiert }`
	`  Writeln('Momentaner Videomodus: ', p^);`
	`end.`

Random-Funktion

Funktion	liefert eine »Zufallszahl« zurück.
Deklaration	`Random [ (Range: Word) ]`
Ergebnistyp	*Real* oder *Word*, abhängig von der Art des Aufrufs.
Erläuterung	wenn *Range* nicht angegeben ist, liefert *Random* einen Realwert zurück, der die Bedingung *0 <= Ergebnis < 1* erfüllt. Wenn *Random* dagegen einen Integer-Ausdruck als *Range* übergeben bekommt, dann liefert diese Funktion einen positiven Integer zurück, der die Bedingung *0 <= Ergebnis < Range* erfüllt. Ein Aufruf von *Random* mit dem Wert 0 für *Range* hat immer das Ergebnis 0. Damit *Random* nicht bei jedem Programmlauf dieselbe Folge von »Zufallszahlen« liefert, sollte der Zufallszahlengenerator mit *Randomize* zu Beginn des Programms initialisiert werden.
Querverweis	*Randomize*

Randomize-Prozedur

Funktion	initialisiert den eingebauten »Zufallszahlen«-Generator.
Deklaration	`Randomize`
Erläuterung	*Randomize* verwendet Datum und Uhrzeit des Systems, um einen Startwert für folgende Aufrufe von *Random* zu setzen. Programme, die ohne *Randomize* arbeiten, erzeugen bei jedem Start immer wieder dieselbe Folge von »Zufallszahlen«. *Hinweis:* Der Startwert des Generators ist in einer globalen *LongInt*-Variablen namens *RandSeed* gespeichert durch direkte Zuweisungen an diese Variable läßt sich eine Folge von »Zufallszahlen« beliebig oft wiederholen (z.B. für Testzwecke und Verschlüsselungsprogramme).
Querverweis	*Random*

ReadBuf-Funktion **WinCrt**

Funktion	liest eine Zeile aus einem CRT-Fenster.
Deklaration	`ReadBuf(Buffer: PChar; Count: Word)`
Ergebnistyp	Word
Erläuterung	*Buffer* zeigt auf einen Zeilenpuffer, der für bis zu der mit *Count* definierten Anzahl von Zeichen Platz bietet. Es können bis zu *Count-2* Zeichen eingegeben werden. Wenn der Benutzer [En-

ter] drückt, wird an die Zeile automatisch eine Zeilenendemarkierung (#13 gefolgt von #10) angehängt. Wenn *CheckEOF* auf *True* gesetzt ist, kann der Benutzer die Eingabe auch mit Ctrl+Z abbrechen. Der Zeile wird dann eine Dateiendemarkierung (#26) angehängt. Der gelieferte Wert ist die Anzahl der gelesenen Zeichen inklusive der Dateiendemarkierung.

Querverweis *ReadChar*

Read-Prozedur

Funktion liest eine oder mehrere Komponenten aus einer typisierten Datei bzw. einen oder mehrere Werte aus einer Textdatei in die angegebenen Variablen.

Deklaration
```
Read(var f,v1 [,v2, ...,vn])      für typisierte Dateien
Read([var f: Text;] v1 [,v2,...,vn])      für Textdateien
```

Erläuterung *Read für typisierte Dateien*: *f* steht für eine Datei-Variable beliebigen Typs (außer *Text*), jede der angegebenen Variablen *v* hat denselben Typ wie die Komponenten dieser Datei. Der Positionszeiger innerhalb der Datei wird durch jede Leseaktion um eine Komponente in Richtung auf das Dateiende bewegt. Der Versuch, über das Ende der Datei »hinauszulesen« (d.h. der Aufruf von *Read*, wenn *Eof(f)* den Wert *True* ergibt), erzeugt einen Laufzeitfehler.

Wenn *Read* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Read für Textdateien: wenn *f* angegeben ist, muß dieser Parameter für eine Textdatei-Variable stehen ist *f* nicht angegeben, dann liest *Read* von der Standard-Eingabe (*Input*).

Für die als *v* angegebenen Variablen sind die Typen *Char*, *String* sowie sämtliche Integer- und Real-Typen zulässig, die Mischung mehrerer Typen in einem Aufruf von *Read* ist erlaubt.

Abhängig vom Typ der angegebenen Variablen interpretiert *Read* gelesene Daten auf unterschiedliche Art und Weise:

Char-Variablen: *Read* liest ein Zeichen aus der Datei und ordnet es der angegebenen Variablen zu. Wenn *Eof(f)* vor dem Aufruf von *Read* den Wert *True* ergibt (d.h. das Dateiende erreicht ist), dann bekommt die *Char*-Variable das Zeichen *chr(26)* (= [Ctrl][Z]) zugewiesen. Wenn *Eoln(f)* vor dem Aufruf von *Read* den Wert *True* ergibt (d.h. ein Zeilenende erreicht ist), dann bekommt die *Char*-Variable das Zeichen chr(13) (= Zeilenvorschub) zugewiesen.

Der Positionszeiger wird um ein Zeichen in Richtung auf das Dateiende versetzt aufeinanderfolgende Aufrufe von *Read* mit *Char*-Variablen lesen also die Datei Zeichen für Zeichen.

Integer-Variablen: *Read* erwartet eine Folge numerischer Zeichen, die den in Kapitel 1 »Sprachelemente und Konstanten« des Programmierhandbuches angegebenen Regeln entspricht. Führende Leerzeichen, Tabs und Zeilenvorschübe werden automatisch übersprungen, das erste nicht-numerische Zeichen (oder das Erreichen des Dateiendes) bricht den Lesevorgang ab.

Wenn die Interpretation der gelesenen Zeichenfolge einen Wert des erwarteten Formats (*Byte*, *ShortInt*, *Integer* usw.) ergibt, wird dieser Wert der angegebenen Variablen zugewiesen, ansonsten erzeugt *Read* einen Laufzeitfehler.

Wenn das Dateiende vor dem Aufruf von *Read* bereits erreicht war oder *Eof(f)* während des Überspringens führender Leerzeichen, Tabs und Zeilenvorschübe den Wert *True* annimmt, weist *Read* der angegebenen Variablen den Wert 0 zu.

Der Positionszeiger bleibt nach der Leseaktion auf dem Zeichen stehen, durch das die Leseaktion abgebrochen wurde der nächste Aufruf von *Read* liest also dieses Zeichen als erstes und überspringt es, wenn wiederum ein numerischer Wert erwartet wird.

Real-Variablen: *Read* erwartet eine Folge numerischer Zeichen, die den in Kapitel 1 »Sprachelemente und Konstanten« des Programmierhandbuches definierten Regeln entspricht. Im Unterschied zu Integerwerten stellen der Dezimalpunkt und das »E« für einen Exponenten hier gültige Zeichen dar, hexadezimale Ziffern dagegen nicht. Führende Leerzeichen, Tabs und Zeilenvorschübe werden automatisch übersprungen, das erste nicht-numerische Zeichen (oder das Erreichen des Dateiendes) bricht den Lesevorgang ab.

Wenn die Interpretation der gelesenen Zeichenfolge einen Wert des erwarteten Formats (*Byte*, *ShortInt*, *Integer* usw.) ergibt, wird dieser Wert der angegebenen Variablen zugewiesen, ansonsten erzeugt *Read* einen Laufzeitfehler.

Wenn das Dateiende vor dem Aufruf von *Read* bereits erreicht war oder *Eof(f)* während des Überspringens führender Leerzeichen, Tabs und Zeilenvorschübe den Wert *True* annimmt, weist *Read* der angegebenen Variablen den Wert 0.0 zu.

Der Positionszeiger bleibt nach der Leseaktion auf dem Zeichen stehen, durch das die Leseaktion abgebrochen wurde der nächste Aufruf von *Read* liest also dieses Zeichen als erstes und überspringt es, wenn wiederum ein numerischer Wert erwartet wird.

String-Variablen: *Read* liest solange Zeichen ab der momentanen Position, bis *Eoln(f)* den Wert *True* annimmt (d.h. ein Zeilenende erreicht ist). Sämtliche gelesenen Zeichen, außer dem Zeilenende selbst, werden der String-Variablen zugeordnet.

Wenn zwischen der momentanen Position und dem nächsten Zeilenende mehr als 255 Zeichen liegen, liest *Read* dennoch bis zum nächsten Zeilenende (wobei kein Laufzeitfehler erzeugt wird). Die String-Variable bekommt in diesem Fall die *ersten* 255 gelesenen Zeichen zugewiesen, die folgenden Zeichen bis zum Zeilenende werden ignoriert.

Der Positionszeiger innerhalb der Datei bleibt nach der Leseaktion auf dem Zeilenende stehen der nächste Aufruf von *Read* liest also das Zeilenende als erstes Zeichen.

Hinweis: Da *Read* Zeilenenden *nicht* automatisch überspringt, brechen aufeinanderfolgende Leseaktionen mit String-Variablen ab dem zweiten Aufruf immer wieder am selben Zeilenende ab und liefern einen Nullstring zurück. Zum Lesen aufeinanderfolgender Textzeilen muß deshalb die Prozedur *Readln* verwendet werden.

Wenn *Read* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Grenzen — die durch *f* angegebene Datei muß für Leseoperationen geöffnet worden sein.

Querverweis — *Readln*, *ReadKey*, *Write*, *Writeln*

ReadKey-Funktion — WinCrt

Funktion — liest ein Zeichen von der Tastatur, ohne es automatisch auf den Bildschirm auszugeben.

Deklaration — `ReadKey`

Ergebnistyp — *Char*

Erläuterung — Das Zeichen wird nicht auf dem Bildschirm angezeigt. Die Funktion *ReadKey* unterstützt nur ASCII-Tastaturcodes, nicht dagegen erweiterte Tastaturcodes wie zum Beispiel Funktions- oder Cursortastencodes.

Readln-Prozedur

Funktion führt einen Aufruf von *Read* aus und springt dann zum Anfang der nächsten Zeile innerhalb der angegebenen Datei.

Deklaration
```
Readln( [var f: Text;] v1 [,v2,...,vn])
```

Erläuterung *Readln* ist eine Erweiterung der Prozedur *Read*, die sich nur auf Textdateien anwenden läßt.

Aufrufe der Form *Readln* und *Readln(f)* d.h. ohne Angaben von Ziel-Variablen führen eine Leseaktion bis zum nächsten Zeilenende durch (oder dem Ende der Datei, falls kein Zeilenende gefunden wird).

Readln ohne Parameterliste entspricht *Readln*(Input).

Wenn *Readln* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Grenzen *Readln* kann (im Gegensatz zu *Read*) nur auf Textdateien (inklusive der Standard-Eingabe) angewendet werden. Die entsprechende Datei muß für Leseoperationen geöffnet worden sein.

Querverweis *Read*

RemoveDir-Prozedur **WinDos**

Funktion entfernt leere Unterverzeichnisse.

Deklaration
```
Remove (Dir: PChar)
```

Erläuterung Das Unterverzeichnis mit dem in *Dir* angegegebenen Pfad wird entfernt. Fehler, wie zum Beispiel ein nicht existierendes Unterverzeichnis oder ein Unterverzeichnis, das nicht leer ist, werden in der Variable *DosError* angegeben.

Querverweis *GetCurDir, CreateDir, SetCurDir. RmDir* entfernt ebenfalls ein leeres Unterverzeichnis, es wird dazu aber ein Pascal-String als Argument benötigt, nicht ein null-terminierter String.

Rename-Prozedur

Funktion gibt einer externen Datei einen neuen Namen.

Deklaration
```
Rename(var f; newname)
```

Erläuterung *f* ist eine Datei-Variable beliebigen Typs.

newname ist ein String-Ausdruck oder ein PChar-Ausdruck, wenn die erweiterte Syntax aktiviert ist. Sie gibt den neuen

Namen der Datei an. Weitere Operationen über *f* werden an der externen Datei mit dem neuen Namen durchgeführt.

Wenn *Rename* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Grenzen *Rename* darf unter keinen Umständen auf eine offene Datei angewendet werden.

Querverweis *Erase*

Reset-Prozedur

Funktion öffnet eine existierende Datei.

Deklaration `Reset(var f [: file; recsize: Word] )`

Erläuterung *f* steht für eine Datei-Variable beliebigen Typs, der zuvor mit *Assign* eine externe Datei zugeordnet wurde. Der optionale Parameter *recsize* ist ein Ausdruck des Typs *Word*, legt die Recordgröße fest und kann nur auf untypisierte Dateien angewendet werden.

Reset öffnet die Datei mit dem bei *Assign* angegebenen Namen bzw. bricht mit einem Laufzeitfehler ab, wenn keine Datei des angegebenen Namens existiert (*Assign* selbst nimmt keine Prüfungen vor). Wenn die durch *f* bezeichnete Datei zum Zeitpunkt des Aufrufs von *Reset* bereits offen ist, wird sie zuerst geschlossen und danach wieder geöffnet. In beiden Fällen setzt *Reset* den Positionszeiger innerhalb dieser Datei auf den Dateianfang.

Wenn *f* mit *Assign* ein Nullstring als Name zugeordnet wurde (*Assign(f, '')*), dann bezieht sich *f* nach dem Aufruf von *Reset* auf die Standard-Eingabe (*Input*, Standard-Handle 0).

Wenn *f* als Textdatei-Variable deklariert ist, eröffnet *Reset* die damit verbundene Datei ausschließlich für Leseoperationen. Falls die Datei leer ist (0 Bytes Umfang), dann liefert *Eof(f)* bereits unmittelbar nach der Eröffnung mit *Reset* den Wert *True* zurück; ansonsten ist das Ergebnis dieser Funktion *False*.

Wenn *f* als untypisierte Datei-Variable deklariert ist, dann öffnet *Reset* die damit verbundene Datei für Lese- und Schreiboperationen, die Recordgröße der Datei kann über den optionalen Parameter *recsize* bei der Eröffnung festgelegt werden. Ist dieser Parameter nicht angegeben, dann setzt *Reset* eine Recordlänge von 128 als Standardvorgabe.

Wenn *Reset* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Querverweis *Append, Assign, Rewrite, Truncate*

Beispiel
```
program ResetDemo;
function Datei_vorhanden (Dateiname: string) : Boolean;
var f: file;
begin
  Assign(f, Dateiname);
  {$I-} Reset(f); {$I+}
  if IOResult = 0 then begin
     Close(f);
     Datei_vorhanden := Dateiname <> '';
   end else
     Datei_vorhanden := False;
end;
begin
  { Dateiname wird als 1. Kommandozeilen-
    Parameter angenommen }
  write('Datei existiert');
  if not Datei_vorhanden(ParamStr(1))
    then write(' nicht!');
end.
```

Rewrite-Prozedur

Funktion erzeugt und öffnet eine neue Datei.

Deklaration `Rewrite(var f [: file; recsize: Word] )`

Erläuterung *f* steht für eine Datei-Variable beliebigen Typs, der zuvor mit *Assign* eine externe Datei zugeordnet wurde. Der optionale Parameter *recsize* ist ein Ausdruck des Typs *Word*, legt die Recordgröße fest und kann nur auf untypisierte Dateien angewendet werden.

Rewrite erzeugt eine Datei mit dem bei *Assign* angegebenen Namen. Wenn die durch *f* bezeichnete Datei zum Zeitpunkt des Aufrufs von *Rewrite* bereits existiert, wird sie vollständig gelöscht und danach neu erzeugt. Dasselbe ist der Fall, wenn die Datei zum Zeitpunkt des Aufrufs von *Rewrite* offen ist. In allen drei Fällen setzt *Reset* den Positionszeiger innerhalb dieser Datei auf den Dateianfang.

Wenn *f* mit *Assign* ein Nullstring als Name zugeordnet wurde (*Assign(f, '')*), dann bezieht sich *f* nach dem Aufruf von *Rewrite* auf die Standard-Ausgabe (*Output*, DOS-Handle 1).

Wenn *f* als Textdatei-Variable deklariert ist, öffnet *Rewrite* die damit verbundene Datei ausschließlich für Schreiboperationen.

Wenn *f* als untypisierte Datei-Variable deklariert ist, dann öffnet *Rewrite* die damit verbundene Datei für Lese- und Schreiboperationen, die Recordgröße der Datei kann über den optio-

nalen Parameter *recsize* bei der Öffnung festgelegt werden. Ist dieser Parameter nicht angegeben, dann setzt *Reset* eine Recordlänge von 128 als Standardvorgabe.

Wenn *Rewrite* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Querverweis *Append*, *Assign*, *Reset*, *Truncate*

Beispiel

```
var
  f : Text;
begin
  Assign(f, 'NEUTEXT.TXT');
  Rewrite(f);
  Writeln(f, 'Gerade eine neue Datei mit einem ',
             'recht nichtssagenden Inhalt erzeugt...');
  Close(f);
end.
```

RmDir-Prozedur

Funktion löscht ein leeres Unterverzeichnis.

Deklaration `RmDir(s: string)`

Erläuterung *s* ist ein String-Ausdruck, gibt den Namen des zu löschenden Unterverzeichnisses an und kann optional einen Laufwerksbezeichner und/oder einen Suchweg enthalten.

Wenn der angegebene Suchweg nicht existiert, das zu löschende Verzeichnis noch Dateieinträge enthält oder momentan als Standardverzeichnis gesetzt ist, dann erzeugt *RmDir* einen I/O-Fehler.

Wenn *RmDir* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Querverweis *ChDir*, *GetDir*, *MkDir*

Beispiel

```
program RmDirDemo;
begin
  { Name des Verzeichnisses wird als
    1. Kommandozeilen-Parameter vorausgesetzt }
  {$I-} RmDir(ParamStr(1)); {$I+}
  if IOResult = 0 then
    writeln('Verzeichnis gelöscht!')
   else
    writeln('... kann nicht gelöscht werden.');
end.
```

Round-Funktion

Funktion rundet das (Real-)Argument auf einen ganzzahligen Wert.

Deklaration
```
Round(x: Real);
```

Ergebnistyp *LongInt*

Erläuterung *x* ist ein Real-Ausdruck, der von *Round* auf den nächstliegenden ganzzahligen Wert gerundet und danach in einen *LongInt* konvertiert wird. Wenn sich der Wert von *x* exakt zwischen zwei ganzen Zahlen befindet, wird in Richtung des größeren Absolutbetrags gerundet: *Round(-3.5)* ergibt also *-4*, *Round(3.5)* ergibt *4*.

Wenn der Wert von *x* außerhalb des zulässigen Rechenbereichs von *LongInt* liegt, erzeugt *Round* einen Laufzeitfehler.

Querverweis *Int*, *Trunc*

RunError-Prozedur

Funktion erzeugt einen Laufzeitfehler, der das Programm definiert abbricht.

Deklaration
```
RunError [(ErrorCode: Word) ]
```

Erläuterung *RunError* entspricht in seiner Wirkungsweise der Standardprozedur *Halt* abgesehen davon, daß diese Routine zusätzlich die Ausgabe einer Laufzeitfehlermeldung veranlaßt (wobei die Adresse des Punktes angegeben wird, von dem aus *RunError* aufgerufen wurde).

Der Integer-Ausdruck *ErrorCode* ist optional und legt die auszugebende Fehlernummer fest wenn er fehlt, setzt der Compiler hier den Wert 0 ein.

Diese Routine ist vor allem für das Austesten von Exit-Prozeduren gedacht, die ein Programm im Fehlerfall »ordentlich« abschließen sollen.

Hinweis: Wenn das Modul, das den Aufruf von *RunError* enthält, mit *Debug information..On* compiliert wurde, reagiert die integrierte Entwicklungsumgebung exakt so, als ob ein echter Laufzeitfehler stattgefunden hätte (d.h. mit der Lokalisierung der »Fehlerstelle« und der Ausgabe einer entsprechenden Meldung).

Querverweis *Halt*, *Exit*

Beispiel
```
....
{$IFDEF DEBUG}   { <- Ein Symbol, das für den Test des
                     Programms definiert wurde }
   if P = nil then RunError(204);
{$ENDIF}
```

ScrollTo-Prozedur **WinCrt**

Funktion positioniert den Inhalt des Fensters so, daß der über die virtuellen Koordinaten X und Y angegebene Bereich ab der linken oberen Ecke des Fensters dargestellt wird.

Deklaration
```
ScrollTo(X, Y: Integer)
```

Erläuterung Die Position der Koordinaten (0,0) entspricht der linken oberen Ecke des virtuellen Bildschirms.

Seek-Prozedur

Funktion setzt den Positionszeiger in einer Datei auf eine bestimmte Komponente.

Deklaration
```
Seek(var f; n: LongInt)
```

Erläuterung *f* steht für eine Datei-Variable beliebigen Typs (außer *Text*), *n* ist ein Ausdruck vom Typ *LongInt* und gibt die Nummer der Komponente an, auf die der Positionszeiger innerhalb der Datei gesetzt werden soll.

Die Zählung der Komponenten einer Datei beginnt mit 0 *Seek(f,0)* setzt den Positionszeiger also an den Dateianfang. Damit sich eine Datei durch nachfolgende Schreiboperationen vergrößern läßt, ist der Aufruf *Seek(f,FileSize(f))* definiert: er setzt den Positionszeiger *hinter* die letzte Komponente.

Wenn *Seek* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Grenzen *Seek* setzt voraus, daß die durch *f* bezeichnete Datei offen ist, und kann nicht auf Textdateien angewendet werden.

Querverweis *FilePos*

SeekEof-Funktion

Funktion prüft, ob sich zwischen der momentanen Position und dem Ende einer Datei noch »lesbare« Daten befinden.

Deklaration
```
SeekEof [ (var f: Text) ]
```

Ergebnistyp *Boolean*

Erläuterung *SeekEof* folgt denselben Regeln wie *Eof* mit einem Unterschied: *Eof* bewegt den Positionszeiger innerhalb der Datei nicht und liefert nur dann *True*, wenn das Dateiende bereits vor dem Aufruf der Funktion erreicht wurde. *SeekEof* liest dagegen so lange Zeichen ab der momentanen Position, bis entweder ein

»lesbares« Zeichen (kein Leerzeichen, Tab oder Zeilenvorschub) oder das Dateiende erreicht ist erst dann wird die eigentliche Prüfung ausgeführt und ein entsprechendes Ergebnis zurückgeliefert. (Sinnvoll ist *SeekEof* vor allem beim Lesen numerischer Daten).

Wenn *SeekEof* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Grenzen *SeekEof* setzt voraus, daß *f* eine offene Textdatei bezeichnet.

Querverweis *Eof*, *SeekEoln*

SeekEoln-Funktion

Funktion prüft, ob sich zwischen der momentanen Position und dem nächsten Zeilenende einer Datei noch »lesbare« Daten befinden.

Deklaration `SeekEoln [ (var f: Text) ]`

Ergebnistyp *Boolean*

Erläuterung *SeekEoln* folgt denselben Regeln wie *Eoln* mit einem Unterschied: *Eoln* bewegt den Positionszeiger innerhalb der Datei nicht und liefert nur dann *True*, wenn ein Zeilenende bereits vor dem Aufruf der Funktion erreicht wurde. *SeekEoln* liest dagegen solange Zeichen ab der momentanen Position, bis entweder ein »lesbares« Zeichen (kein Leerzeichen oder Tab) oder ein Zeilenende (bzw. das Dateiende) erreicht ist erst dann wird die eigentliche Prüfung ausgeführt und ein entsprechendes Ergebnis zurückgeliefert.

Sinnvoll ist *SeekEoln* vor allem beim Lesen von numerischen Daten.

Wenn *SeekEoln* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Grenzen *SeekEoln* setzt voraus, daß *f* eine offene Textdatei bezeichnet.

Querverweis *Eoln*, *SeekEof*

Seg-Funktion

Funktion	liefert die Segmentadresse des angegebenen Objekts zurück.
Deklaration	`Seg(x)`
Ergebnistyp	*Word*
Erläuterung	*x* steht für eine beliebige Variable, einen beliebigen Variablenbezug oder den Bezeichner einer Prozedur oder Funktion. Das Ergebnis von *Seg* ist die Adresse des Segments, in dem *x* gespeichert ist.
Querverweis	*Addr*, *Ofs*

SetCBreak-Prozedur **WinDos**

Funktion	legt fest, bei welchen Operationen DOS auf `[Ctrl][Break]` prüft.
Deklaration	`SetCBReak(Break: Boolean)`
Erläuterung	*SetCBreak* setzt das *Break*-Flag von DOS (über die DOS-Funktion $33) mit dem Wert des als *Break* übergebenen Ausdrucks:

True	Überprüfung an DOS prüft bei jedem Funktionsaufruf, ob `[Ctrl][Break]` gedrückt wurde;
False	Überprüfung aus die Prüfung auf `[Ctrl][Break]` findet nur bei Ein- und Ausgaben über Tastatur, Bildschirm, serielle Schnittstellen und Drucker statt.

Hinweis: Mit der Prozedur *GetCBreak* läßt sich der momentane Wert des *Break*-Flags ermitteln.

Querverweis *GetCBreak*

SetCurDir-Prozedur **WinDos**

Funktion	ändert das aktuelle Verzeichnis zu dem Pfad, der mit *Dir* angegeben wird.
Deklaration	`SetCurDir(Dir: PChar)`
Ergebnistyp	PChar
Erläuterung	Wenn *Dir* einen Laufwerksnamen angegeben hat, wird auch das aktuelle Laufwerk geändert. Fehler werden in *DosError* angegeben.
Querverweis	*GetCurDir, CreateDir, RemoveDir. ChDir* hat dieselbe Funktion wie *SetCurDir*, es wird dazu aber ein Pascal-String als Argument benötigt, kein null-terminierter String.

SetDate-Prozedur WinDos

Funktion setzt das Kalenderdatum des Betriebssystems.

Deklaration
```
SetDate(Year, Month, Day, DayofWeek: Word)
```

Erläuterung gültige Werte für *Year* gehen von 1980..2099, für *Month* von 1..12, für *Day* von 1..31 und für *DayofWeek* (Wochentag) von 0..6, wobei 0 für den Sonntag steht.

Querverweis *GetTime*, *GetDate*, *SetTime*

SetFAttr-Prozedur WinDos

Funktion setzt die Attribute einer Datei.

Deklaration
```
SetFAttr(var f; Attr: Byte)
```

Erläuterung *f* steht für eine Datei-Variable beliebigen Typs, der über *Assign* der Name einer externen Diskettendatei zugeordnet wurde. Die Datei darf nicht offen sein.

Attr ist ein Ausdruck des Typs *Byte*, dessen Wert sich aus der Addition mehrerer in der Unit *WinDos* definierter Konstanten ergibt:

```
const
  faReadOnly  = $01;
  faHidden    = $02;
  faSysFile   = $04;
  faVolumeID  = $08;
  faDirectory = $10;
  faArchive   = $20;
  faAnyFile   = $3F;
```

Fehler während der Ausführung von *SetFAttr* setzen die globale Variable *DosError*, mögliche Codes sind hier 3 (»ungültiger Suchweg«) und 5 (»Dateizugriff verweigert«).

Grenzen die durch *f* bezeichnete Datei darf nicht offen sein.

Querverweis *GetFAttr*, *GetFTime*, *SetFTime*

Beispiel
```
program SetFAttrDemo;
   { Quelltext: SETFATT.PAS }
uses WinDos;
var
  f : file;
begin
  Assign(f, 'SETFATT.PAS');
  SetFAttr(f, ReadOnly + Hidden + SysFile);
  Writeln('So. Und jetzt dürfen Sie mal versuchen.');
```

```
  Writeln('diese Datei wieder zu löschen.');
end.
```

SetFTime-Prozedur **WinDos**

Funktion setzt Datum und Uhrzeit der letzten Veränderung einer Datei direkt.

Deklaration `SetFTime(var f; Time: LongInt)`

Erläuterung *f* steht für eine Datei-Variable beliebigen Typs, die entsprechende Datei muß offen sein. Der *LongInt*-Ausdruck *Time* enthält Datum und Uhrzeit in einem speziellen gepackten Format, das sich am einfachsten durch die Anwendung der Prozedur *PackTime* erzeugen läßt.

Fehler bei der Ausführung von *SetFTime* setzen die globale Variable *DosError*, der einzig mögliche Code ist hier 6 (»ungültiger Handle/Datei nicht offen«).

Grenzen die durch *f* bezeichnete Datei muß offen sein.

Querverweis *GetFTime*, *PackTime*, *UnpackTime*

SetIntVec-Prozedur **WinDos**

Funktion setzt einen Interrupt-Vektor des Systems auf eine bestimmte Adresse.

Deklaration `SetIntVec(IntNo: Byte; Vector: Pointer)`

Erläuterung *IntNo* ist ein Ausdruck des Typs *Byte* und gibt die Nummer des Vektors an (0..255), der auf die durch *Vector* festgelegte Adresse gesetzt werden soll.

Vector läßt sich entweder über die Funktion *Addr*, den Adress-Operator *@* oder über *Ptr* erzeugen.

Unter der Annahme, daß *Int1BSave* eine Variable vom Typ Pointer und *Init1BHandler* ein Interrupt-Prozedur-Bezeichner ist, installiert folgende Anweisung einen neuen Interrupt-Handler $1B und stellt später den ursprünglichen Handler wieder her:

```
GetIntVec($1B, Int1BSave);
SetIntVec($1B; @Int1BHandler);
...
SetIntVec($1B, Int1BSave);
```

Querverweis *GetIntVec*

SetTextBuf-Prozedur

Funktion weist einer Textdatei-Variablen einen Puffer zu.

Deklaration
```
SetTextBuf(var f: Text; var buf [; size: Word] )
```

Erläuterung *f* ist eine Textdatei-Variable, *buf* ist eine Variable beliebigen Typs, *size* ein optionaler *Word*-Ausdruck.

Jede Textdatei-Variable verfügt über einen eigenen Puffer von 128 Bytes Größe, der standardmäßig von *Read* und *Write* benutzt wird. Für die meisten Anwendungen sind Puffer dieser Größe ausreichend; falls Sie allerdings ganze Textdateien zeilenweise kopieren wollen oder andere ähnlich aufwendige Datei-Operationen vorhaben, sollte der Puffer auf einige KByte vergrößert werden. Das Programm wird durch größere Dateipuffer schneller.

SetTextBuf verändert die Dateivariable *f* so, daß die als *buf* angegebene Variable zur Zwischenspeicherung benutzt wird. Wenn der Parameter *size* angegeben ist, wird der entsprechende Teil von *buf* benutzt; ansonsten setzt der Compiler anstelle von *size* automatisch die Größe der Variablen *buf* ein (also *SizeOf(buf)*).

Als Voreinstellung wird der gesamte von *buf* besetzte Speicherbereich als Puffer verwendet.

Grenzen *SetTextBuf* sollte nie auf eine offene Datei angewendet werden da die Puffer-Variable einfach ausgetauscht wird, gehen sonst Daten verloren (ein Aufruf *direkt* nach einem *Reset*, *Rewrite* oder *Append* ist allerdings möglich). *f* muß eine Textdatei bezeichnen.

Vorsicht! Turbo Pascal kann nicht bestimmen, ob eine als Puffer zugeordnete Variable diesen Zweck auch während der gesamten Laufzeit des Programms erfüllen kann. Einer der häufigsten Fehler in diesem Zusammenhang ist die Deklaration der Puffervariablen innerhalb einer lokalen Routine.

Beispiel
```
program SetTextBufDemo;
var
  f : Text;
  ch : Char;
  buf: array[1..10240] of Char;  { 10 KByte }
begin
  { Dateiname wird als 1. Kommandozeilen-
    Parameter angenommen }
  Assign(f, ParamStr(1));
  SetTextBuf(f, buf); { Zuordnung des größeren Puffers }
  Reset(f);
  { Textdatei auf dem Bildschirm ausgeben }
  while not Eof(f) do
  begin
```

```
    Read(f, ch);
    Write(ch);
  end;
  Writeln;  Close(f);
end.
```

SetTime-Prozedur **WinDos**

Funktion	setzt die Uhrzeit des Systems.
Deklaration	`SetTime(Hour, Minute, Second, Sec100: Word)`
Erläuterung	gültige Werte für *Hour* gehen von 0..23, für *Minute* und *Second* von 0..59 und für *Sec100* (hundertstel Sekunden) von 0..99. Wenn auch nur einer der Parameter außerhalb dieser Grenzen liegt, bleibt die Uhrzeit des Systems unverändert.
Querverweis	*GetDate*, *GetTime*, *SetDate*

SetVerify-Prozedur **WinDos**

Funktion	setzt das *Verify*-Flag von DOS.
Deklaration	`SetVerify(Verify: Boolean)`
Erläuterung	das Flag *Verify* legt fest, ob DOS Diskettensektoren nach dem Schreiben (durch eine reine Leseaktion ohne direkten Datenvergleich) noch einmal prüft oder nicht. Die Prozedur *SetVerify* setzt dieses Flag mit dem als *Verify* übergebenen Ausdruck: *True* – DOS führt eine zusätzliche Überprüfung aus. *False* – Diskettensektoren werden nach dem Schreiben nicht noch einmal zur Verifizierung der Prüfsumme gelesen.
Querverweis	*GetVerify*

Sin-Funktion

Funktion	liefert den Sinus des Arguments zurück.
Deklaration	`Sin(x: Real)`
Ergebnistyp	*Real*
Erläuterung	*x* ist ein Real-Ausdruck und stellt einen Winkel in der Einheit *Rad* (0..2 Pi) dar. Das Ergebnis der Funktion ist der Sinus von *x*.
Querverweis	*ArcTan*, *Cos*
Beispiel	`var  x : Real;` `begin`

```
  x := Sin(Pi);   { 0 }
end.
```

SizeOf-Funktion

Funktion — liefert die Anzahl von Bytes zurück, die das Argument an Speicherplatz belegt.

Deklaration — `SizeOf(x)`

Ergebnistyp — *Word*

Erläuterung — *x* ist entweder ein beliebiger Variablenbezug oder ein Typ-Bezeichner. Das Ergebnis der Funktion ist der Platzbedarf von *x* in Bytes.

Wird diese Funktion auf eine Instanz eines Objekttyps mit einer virtuellen Methodentabelle (VMT) angewendet, dann liefert *SizeOf* die Größe, die in der VMT gespeichert ist. Damit erhält man also für Objekttypen, die eine VMT haben, die *tatsächliche* Größe der Instanz und nicht die *deklarierte* Größe.

Zur Angabe von Bereichsgrößen als Parameter von *FillChar, Move, GetMem* usw. sollte *SizeOf* anstelle von Konstanten verwendet werden:

```
FillChar(s, SizeOf(s), 0);
GetMem(p, SizeOf(Record_Typ));
```

Beispiel

```
type
  Eintrag = record
     Name   : string[30];
     Telefon: string[14];
    end;
var
  p : ^Eintrag;
begin
  GetMem(p, SizeOf(Eintrag));
  ...
end.
```

SPtr-Funktion

Funktion — liefert den momentanen Wert des Stackzeigers (SP-Register) zurück.

Deklaration — `SPtr`

Ergebnistyp — *Word*

Erläuterung — das Ergebnis von *SPtr* entspricht dem *Offset*-Anteil der Spitze des Stacks, d.h. dem Stackzeiger innerhalb des Stack-Segments.

Querverweis *SSeg*

Sqr-Funktion

Funktion liefert das Quadrat des Arguments zurück.

Deklaration `Sqr(x)`

Ergebnistyp derselbe Typ wie der des Arguments.

Erläuterung *x* ist entweder ein Integer- oder ein Real-Ausdruck. Das Ergebnis der Funktion hat denselben Typ wie *x* und entspricht der Berechnung *x* * *x*.

Sqrt-Funktion

Funktion liefert die Quadratwurzel (»square root«) des Arguments zurück.

Deklaration `Sqrt(x: Real)`

Ergebnistyp *Real*

Erläuterung x ist ein Ausdruck des Typs *Real*. Das Ergebnis der Funktion ist die Quadratwurzel von *x*.

Grenzen negative Werte von *x* sind mathematisch nicht definiert und erzeugen einen Laufzeitfehler.

SSeg-Funktion

Funktion liefert die Adresse des Stack-Segments zurück.

Deklaration `SSeg`

Ergebnistyp *Word*

Erläuterung das Ergebnis der Funktion ist der Inhalt des Prozessor-Registers SS.

Querverweis *CSeg*, *DSeg*, *SPtr*

Str-Prozedur

Funktion konvertiert einen numerischen Wert in einen String.

Deklaration `Str(x [ : width [ : decimals ] ], var s)`

Erläuterung *x* stellt den zu konvertierenden Wert dar und ist ein Integer- oder Real-Ausdruck. *Str* bildet einen String, der dieselbe Folge von Zeichen enthält, wie sie *Write* (ohne Angaben von *Breite* und *MinStellen*, siehe »Write-Prozedur) ausgeben würde tat-

sächlich verwenden beide dieselbe Routine der Laufzeitbibliothek zur Erzeugung numerischer Zeichenfolgen. Der Hauptunterschied liegt darin, daß *Write* die erzeugten Zeichen danach ausgibt, während *Str* sie in einem String als Funktionsergebnis zurückliefert.

Querverweis *Val, Write*

Beispiel
```
function itoa(i : LongInt): string;
  { verwandelt von »Integer« nach »ASCII« }
var
  s : string[11];
begin
  Str(i,s);
  itoa := s;
end;
begin
  Writeln('Die folgenden beiden Ausgaben sollten',
          'wirklich gleich sein:');
  Writeln(-5322, ' <---> ', itoa(-5233));
end.
```

StrCat-Funktion — Strings

Funktion hängt eine Kopie eines Strings an das Ende eines anderen an und liefert den verketteten String.

Deklaration `StrCat(Dest, Source: PChar)`

Ergebnistyp PChar

Erläuterung *StrCat* hängt eine Kopie von *Source* an *Dest* an und liefert *Dest* zurück. *StrCat* führt keine Überprüfung der Läge des Strings durch. Sie müssen selber sicherstellen, daß der Puffer von *Dest* Platz für mindestens *StrLen (Dest)* + *StrLen (Source)* + *1* Zeichen bietet. Wenn eine Längenüberprüfung durchgeführt werden soll, verwenden Sie die Funktion *StrLCat*.

Querverweis *StrLCat*

Beispiel
```
uses Strings, WinCrt;
const
  Turbo: PChar = 'Turbo';
  Pascal: PChar = 'Pascal';
var
  S: array[0..15] of Char;
begin
  StrCopy(S, Turbo);
  StrCat(S,'');
  StrCat(S, Pascal);
```

```
  WriteLn(S);
end.
```

StrComp-Funktion — Strings

Funktion	vergleicht zwei Strings.
Deklaration	`StrComp(Str1, Str2: PChar)`
Ergebnistyp	Integer
Erläuterung	*StrComp* vergleicht *Str1* mit *Str2*. Der zurückgelieferte Wert ist kleiner Null, wenn *Str1 Str2*, 0 wenn *Str1 = Str2*, oder größer 0 wenn *Str1 Str2*.
Querverweis	*StrIComp, StrLComp, StrLIComp*
Beispiel	

```
uses Strings, WinCrt;
var
  C: Integer;
  Result: PChar;
  S1, S2: array[0..79] of Char;
begin
  ReadLn(S1);
  ReadLn(S2);
  C := StrComp(S1, S2);
  if C  0 then Result := 'ist kleiner als ' else
    if C  0 then Result := 'ist größer als ' else
      Result := 'ist gleich';
  WriteLn(S1, Result, S2);
end.
```

StrCopy-Funktion — Strings

Funktion	kopiert einen String in einen anderen.
Deklaration	`StrCopy(Dest, Source: PChar)`
Ergebnistyp	PChar
Erläuterung	*StrCopy* kopiert *Source* in *Dest* und liefert *Dest*. *StrCopy* führt keine Längenüberprüfungen durch. Sie müssen sicherstellen, daß der von *Dest* angegebene Puffer ausreicht, um mindestens *StrLen(Source) + 1* Zeichen unterzubringen. Wenn eine Längenüberprüfung durchgeführt werden soll, verwenden Sie die Funktion *StrLCopy*.
Querverweis	*StrECopy, StrLCopy*
Beispiel	

```
uses Strings, WinCrt;
var
```

```
  S: array[0..15] of Char;
begin
  StrCopy(S, 'Turbo Pascal');
  WriteLn(S);
end.
```

StrDispose-Funktion **Strings**

Funktion entfernt einen String vom Heap.

Deklaration `StrDispose(Str: PChar)`

Erläuterung *StrDispose* entfernt einen String vom Heap, der vorher mit *StrNew* angelegt wurde. Wenn *Str* **nil** ist, hat die Funktion *StrDispose* keine Auswirkungen.

Querverweis *StrNew*

StrECopy-Funktion **Strings**

Funktion kopiert einen String in einen anderen und liefert einen Zeiger auf das Ende des resultierenden Strings.

Deklaration `StrECopy(Dest, Source: PChar)`

Ergebnistyp PChar

Erläuterung *StrECopy* kopiert *Source* in *Dest* und liefert *StrEnd(Dest)*. *StrECopy* führt keine Längenüberprüfungen durch. Sie müssen sicherstellen, daß der von *Dest* angegebene Puffer ausreicht, um mindestens *StrLen(Source) + 1* Zeichen unterzubringen. Verschachtelte Aufrufe an *StrECopy* können verwendet werden, um eine Reihe von Strings zu verketten. Dies wird im folgenden Beispiel demonstriert:

Querverweis *StrCopy, StrEnd*

Beispiel

```
uses Strings, WinCrt;
const
  Turbo := PChar = 'Turbo';
  Pascal:= PChar = 'Pascal'
var
  S: array[0..15] of Char;
begin
  StrECopy(StrECopy(StrECopy(S, 'Turbo',''), Pascal;
  WriteLn(S);
end.
```

StrEnd-Funktion — Strings

Funktion liefert einen Zeiger auf das Ende eines Strings.

Deklaration `StrEnd(Str: PChar)`

Ergebnistyp PChar

Erläuterung *StrEnd* liefert einen Zeiger auf das Null-Zeichen, den nullterminierten String *Str* abschließt.

Querverweis *StrLen*

Beispiel
```
uses Strings, WinCrt;
var
  S:array[0..79] of Char;
begin
  ReadLn(S);
  WriteLn('Stringlänge ist ', StrEnd(S) - S);
end.
```

StrIComp-Funktion — Strings

Funktion vergleicht zwei Strings, ohne Beachtung von Groß- und Kleinschreibung.

Deklaration `StrIComp(Str1, Str2: PChar)`

Ergebnistyp Integer

Erläuterung *StrIComp* vergleicht *Str1* mit *Str2*, ohne die Groß- und Kleinschreibung zu beachten. Der zurückgelieferte Wert hat dieselbe Bedeutung wie der Rückgabewert von *StrComp*.

Querverweis *StrComp, StrLComp, StrLIComp*

StrLCat-Funktion — Strings

Funktion hängt Zeichen aus einem String an das Ende eines anderen an und liefert den verketteten String.

Deklaration `StrLCat(Dest, Source: PChar; MaxLen: Word)`

Ergebnistyp PChar

Erläuterung *StrLCat* hängt maximal *MaxLen - StrLen(Dest)* Zeichen aus *Source* an *Dest* an und liefert *Dest*. Die Standardfunktion *SizeOf* kann verwendet werden, um den Parameter *MaxLen* festzulegen. Sehen Sie dazu folgendes Beispiel:

Querverweis *StrCat*

Beispiel

```
uses Strings, WinCrt;
var
  S: array[0..9] of Char;
begin
  StrLCopy(S, 'Turbo', SizeOf(S) -1);
  StrLCat(S,'', SizeOf(S) - 1);
  StrLCat(S, 'Pascal', SizeOf (S) - 1);
  WriteLn(S);
end.
```

StrLComp-Funktion **Strings**

Funktion vergleicht zwei Strings bis zu einer maximalen Anzahl von Zeichen.

Deklaration `StrLComp(Str1, Str2: PChar; MaxLen: Word)`

Ergebnistyp Integer

Erläuterung *StrLComp* vergleicht *Str1* mit *Str2* bis zur maximalen Anzahl der Zeichen (*MaxLen*). Der zurückgelieferte Wert hat dieselbe Bedeutung wie der Rückgabewert von *StrComp*.

Querverweis *StrComp, StrLIComp, StrIComp*

Beispiel

```
uses Strings, WinCrt;
var
  Result: PChar;
  S1, S2: array[0..79] of Char;
begin
  ReadLn(S1);
  ReadLn(S2);
  if StrLComp (S1, S2, 5) = 0 then
    Result := 'gleich'
  else
    Result := 'verschieden';
end.
```

StrLCopy-Funktion **Strings**

Funktion Kopiert Zeichen von einem String in einen anderen.

Deklaration `StrLCopy(Dest, Source: PChar; MaxLen: Word)`

Ergebnistyp PChar

Erläuterung *StrLCopy* kopiert maximal die mit *MaxLen* definierte Anzahl von Zeichen aus *Source* in *Dest*. *MaxLen* kann mit der Standardfunktion *SizeOf* definiert werden. Sehen Sie dazu folgendes Beispiel:

Querverweis *StrCopy*

Beispiel
```
uses Strings, WinCrt;
var
  S: array[0..9] of Char;
begin
  StrLCopy (S, 'Turbo Pascal', SizeOf (S) -1)
  WriteLn(S);
end.
```

StrLen-Funktion **Strings**

Funktion liefert die Anzahl der Zeichen in *Str*.

Deklaration `StrLen(Str: PChar)`

Ergebnistyp Word

Erläuterung *StrLen* liefert die Anzahl der Zeichen in *Str*. Dabei wird der Null-Terminator nicht gezählt.

Querverweis *StrEnd*

Beispiel
```
uses Strings, WinCrt;
var
  S: array[0..79] of Char;
begin
  ReadLn(S);
  WriteLn('Die Stringlänge ist', StrLen (S));
end.
```

StrLIComp-Funktion **Strings**

Funktion vergleicht zwei Strings bis zur maximalen Anzahl von Zeichen, ohne Beachtung von Groß- und Kleinschreibung.

Deklaration `StrLIComp(Str1, Str2: PChar; MaxLen: Word)`

Ergebnistyp Integer

Erläuterung *StrLIComp* vergleicht *Str1* mit *Str2* bis zur maximalen Anzahl der Zeichen (MaxLen) und ohne Beachtung von Groß- und Kleinschreibung. Der zurückgelieferte Wert hat dieselbe Bedeutung wie der Rückgabewert von *StrComp*.

Querverweis *StrComp, StrLComp, StrIComp*

StrLower-Funktion **Strings**

Funktion Konvertiert einen String zu Kleinbuchstaben.

Deklaration `StrLower(Str: PChar)`

Ergebnistyp PChar

Erläuterung *StrLower* konvertiert *Str* in Kleinbuchstaben und liefert *Str*.

Querverweis *StrUpper*

Beispiel
```
uses Strings, WinCrt;
var
  S: array[0..79] of Char;
begin
  ReadLn(S);
  WriteLn(StrLower(S));
  WriteLn(StrUpper(S));
end.
```

StrMove-Funktion **Strings**

Funktion Kopiert Zeichen aus einem String zu einem anderen.

Deklaration `StrMove(Dest, Source: PChar; Count: Word)`

Ergebnistyp PChar

Erläuterung *StrMove* kopiert exakt die mit *Count* definierte Anzahl von Zeichen aus *Source* zu *Dest* und liefert *Dest*. *Source* und *Dest* können sich überschneiden.

Beispiel
```
{ Lokalisieren eines Strings auf dem Heap}
function StrNew (S: PChar): PChar;
var
  L: Word;
  P: PChar;
begin
  if (S = nil) or (S^ = #0) then StrNew := nil else
  begin
    L := StrLen(S) + 1
    GetMem(P, L);
    StrNew := StrMove(P, S, L);
  end;
end;
{Entfernen eines Strings vom Heap}
procedure StrDispose(S:PChar);
begin
  if S  nil then FreeMem (S, StrLen(S) + 1);
end;
```

StrNew-Funktion **Strings**

Funktion Alloziert einen String auf dem Heap.

Deklaration `StrNew(Str: PChar)`

Ergebnistyp PChar

Erläuterung *StrNew* alloziert eine Kopie von *Str* auf dem Heap. Wenn *Str* **nil** ist oder auf einen leeren String zeigt, liefert StrNew entweder nil und belegt keinen Speicher auf dem Heap. Andernfalls legt *StrNew* eine Kopie von *Str* an, wobei der notwendige Speicher über einen Aufruf der Standardprozedur *GetMem* reserviert wird, und liefert einen Zeiger auf den kopierten String. Der belegte Platz umfaßt *StrLen(Str) + 1* Bytes.

Querverweis *StrDispose*

Beispiel
```
uses Strings, WinCrt;
var
  P: PChar;
  S: array[0..79] of Char;
begin
  ReadLn(S);
  P := StrNew(S);
  WriteLn(P);
  StrDispose(P);
end.
```

StrPas-Funktion **Strings**

Funktion konvertiert einen null-terminierten String in einen Pascal-String.

Deklaration `StrPas(Str: PChar)`

Ergebnistyp PChar

Erläuterung *StrPas* konvertiert *Str* in einen Pascal-String.

Querverweis *StrPCopy*

Beispiel
```
uses Strings, WinCrt;
var
  A: array[0..79] of Char;
  S: string[79];
begin
  ReadLn(A);
  S := StrPas(A);
  WriteLn(S);
end.
```

StrPCopy-Funktion **Strings**

Funktion konvertiert einen Pascal-String in einen null-terminierten String.

Deklaration `StrPCopy(Dest: PChar; Source: String)`

Ergebnistyp PChar

Erläuterung *StrPCopy* konvertiert den Pascal-String *Source* in *Dest* und liefert *Dest*. Sie müssen sicherstellen, daß der von *Dest* bezeichnete Puffer genug Platz für *Length(Source) + 1* Zeichen bietet.

Querverweis *StrCopy*

Beispiel

```
uses Strings, WinCrt;
var
  A: array[0..79] of Char;
  S: string[79];
begin
  ReadLn(S);
  StrPCopy(A, S);
  WriteLn(A);
end.
```

StrPos-Funktion **Strings**

Funktion liefert einen Zeiger auf das erste Vorkommen eines Strings in einem anderen String.

Deklaration `StrPos(Str1, Str2: PChar)`

Ergebnistyp PChar

Erläuterung *StrPos* liefert einen Zeiger auf das erste Vorkommen von *Str2* in *Str1*. Wenn *Str2* nicht in *Str1* vorkommt, liefert *StrPos* **nil**.

Beispiel

```
uses Strings, WinCrt;
var
  P: PChar;
  S: SubStr: array[0..79] of Char;
begin
  ReadLn(S);
  ReadLn(SubStr);
  if P = nil then
    WriteLn('Substring nicht gefunden');
  else
    WriteLn('Substring gefunden bei Index', P-S);
end.
```

StrRScan-Funktion — Strings

Funktion — liefert einen Zeiger auf das letzte Vorkommen eines Zeichens in einem String.

Deklaration
```
StrRScan(Str: PChar; Chr: Char)
```

Ergebnistyp — PChar

Erläuterung — *StrRScan* liefert einen Zeiger auf das letzte Vorkommen von *Chr* in *Str*. Wenn *Chr* nicht in *Str* vorkommt, liefert *StrRScan* **nil**. Der Null-Terminator wird als Teil des Strings angesehen.

Querverweis — *StrScan*

Beispiel
```
{Liefert einen Zeiger zum Namensteil
 eines vollen Pfadnamens}
function NamePart (FileName: PChar): PChar;
var
  P: PChar;
begin
  P := StrRScan (FileName, '\');
  if P = nil then
  begin
    P := StrRScan (FileName, ':');
    if P = nil then P := FileName;
  end;
  NamePart :=P;
end.
```

StrScan-Funktion — Strings

Funktion — liefert einen Zeiger auf das erste Vorkommen eines Zeichens in einem String.

Deklaration
```
StrScan(Str: PChar; Chr: Char)
```

Ergebnistyp — PChar

Erläuterung — *StrScan* liefert einen Zeiger zum ersten Vorkommen von *Chr* in *Str*. Wenn *Chr* nicht in *Str* vorkommt, liefert *StrScan* **nil**. Der Null-Terminator wird als Teil des Strings angesehen.

Querverweis — *StrRScan*

Beispiel
```
{Liefert True, wenn der Dateiname Jokerzeichen enthält}
function HasWildcards (FileName: PChar): Boolean;
begin
  HasWildcards :=(StrScan(FileName, '*') nil) or
    (StrScan(FileName, '?')  nil);
end;
```

StrUpper-Funktion **Strings**

Funktion	konvertiert einen String zu Großbuchstaben.
Deklaration	`StrUpper(Str: PChar)`
Ergebnistyp	PChar
Erläuterung	*StrUpper* konvertiert *Str* in Großbuchstaben und liefert *Str*.
Querverweis	*StrLower*

Beispiel	`uses Strings, WinCrt;`
	`var`
	`  S: array[0..79] of Char;`
	`begin`
	`  ReadLn(S);`
	`  WriteLn(StrUpper(S));`
	`  WriteLn(StrLower(S));`
	`end.`

Succ-Funktion

Funktion	liefert den Nachfolger des Arguments zurück.
Deklaration	`Succ(x)`
Ergebnistyp	derselbe wie der des Arguments.
Erläuterung	*x* ist ein ordinaler Ausdruck beliebigen Typs. Das Ergebnis der Funktion hat denselben Typ und stellt den Nachfolger von *x* dar.
Grenzen	wenn *x* das Element mit dem höchsten ordinalen Wert innerhalb seines Typs darstellt, dann ist der Nachfolger von *x* nicht definiert, und *Succ* erzeugt einen Laufzeitfehler.
Querverweis	*Inc*, *Pred*

Swap-Funktion

Funktion	vertauscht das niederwertige und das höherwertige Byte des Arguments miteinander.
Deklaration	`Swap(x)`
Ergebnistyp	derselbe Typ wie der des Arguments.
Erläuterung	*x* ist ein Ausdruck des Typs *Integer* oder *Word*.
Querverweis	*Hi*, *Lo*
Beispiel	`var` `  x : Word;`

```
begin
  x := Swap($1234);   { x := $3412 }
end.
```

TrackCursor-Procedur **WinCrt**

Funktion	Rollt das CRT-Fenster ab, um sicherzustellen, daß der Cursor sichtbar bleibt.
Deklaration	`TrackCursor;`

Trunc-Funktion

Funktion	wandelt einen Realwert durch Abschneiden der Nachkommastellen in einen Integerwert um.
Deklaration	`Trunc(x: Real)`
Ergebnistyp	*LongInt*
Erläuterung	*x* ist ein Real-Ausdruck. Das Ergebnis der Funktion ist der ganzzahlige Anteil von *x*.
Grenzen	wenn der Wert von *x* außerhalb des Rechenbereichs von *LongInt* liegt, dann erzeugt *Trunc* einen Laufzeitfehler.
Querverweis	*Int*, *Round*

Truncate-Prozedur

Funktion	schneidet eine Datei an der momentanen Position ab.
Deklaration	`Truncate(var f)`
Erläuterung	*f* steht für eine Datei-Variable beliebigen Typs (außer *Text*). Die mit *f* verbundene Datei wird an der Stelle abgeschnitten, auf die der Positionszeiger zum Zeitpunkt des Aufrufs von *Truncate* zeigt. Zwischen dieser Position und dem physikalischen Dateiende befindliche Daten werden gelöscht, die Funktion *Eof(f)* liefert nach der Ausführung von *Truncate* das Ergebnis *True*. Wenn *Truncate* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.
Grenzen	die durch *f* bezeichnete Datei muß offen sein. *Truncate* läßt sich nicht auf Text-Dateien anwenden.
Querverweis	*Seek*, *Reset*, *Rewrite*

UnpackTime-Prozedur WinDos

Funktion konvertiert Datum und Uhrzeit aus einem gepackten Format (4 Byte) in einen Record des Typs *TDateTime*.

Deklaration
```
UnpackTime(Time: LongInt; var DT: TDateTime)
```

Erläuterung »gepackte« *LongInt*-Variablen, die ein Kalenderdatum und eine Uhrzeit zusammen in 4 Bytes speichern, werden von den Routinen *GetFTime* und *PackTime* sowie von *FindFirst* und *FindNext* erzeugt. *UnpackTime* »entpackt« einen solchen *LongInt* und setzt die Felder des als *DT* übergebenen Records mit den einzelnen Werten. Der Datentyp *DateTime* ist in der Unit *WinDos* folgendermaßen definiert:

```
TDateTime = record
    Year, Month,    { Jahr: 1980..2099, Monat 1..12 }
    Day, Hour,      { Tag: 1..31, Stunde 0..23 }
    Min, Sec : Word;  { jeweils 0..59 }
  end;
```

Die einzelnen Bit-Felder des als *Time* übergebenen Wertes werden nicht geprüft.

Querverweis *PackTime*, *GetFTime*, *SetFTime*, *GetTime*, *SetTime*

UpCase-Funktion

Funktion verwandelt Klein- in Großbuchstaben.

Deklaration
```
UpCase(ch: Char)
```

Ergebnistyp *Char*

Erläuterung *ch* ist ein Ausdruck des Typs *Char*. Wenn *ch* einen Kleinbuchstaben im Bereich von 'a'..'z' darstellt, liefert *UpCase* den entsprechenden Großbuchstaben zurück.

Val-Prozedur

Funktion interpretiert einen String als numerischen Wert.

Deklaration
```
Val(s: string; v; var code: Integer)
```

Erläuterung *s* ist der zu interpretierende String-Ausdruck oder ein Ausdruck vom Typ *PChar* wenn die erweiterte Syntax aktiviert ist. *v* ist eine Variable mit einem Integer- oder Real-Typ. In der als *code* übergebenen Integer-Variablen speichert *Val* bei einer erfolgreichen Konvertierung den Wert 0 ansonsten steht hier die Position des Zeichens in *s*, durch das die Interpretation als fehlerhaft abgebrochen wurde.

s muß eine Zeichenfolge enthalten, die den in Kapitel 1

»Sprachelemente und Konstanten« des Programmierhandbuches aufgeführten Regeln genügt.

Val konvertiert *s* in seine numerische Entsprechung und speichert das Ergebnis in *v*. Wenn der String ungültig ist, wird der Index des ungültigen Zeichens in *code* gespeichert. Andernfalls wird *code* auf Null gesetzt. Bei null-terminierten Strings ist die Fehler-Position, die in *code* zurückgeliefert wird, um eine Position größer als der tatsächliche, auf Null basierende Index des Zeichens, das den Fehler auslöste.

Die Fehlerprüfung von *Val* hängt von {$R}-Modus und von dem Typ des Parameters *v* ab. Wenn die Bereichsprüfung eingeschaltet ist ({$R+}), erzeugt ein Wert außerhalb des Bereichs einen Laufzeitfehler. Wenn die Bereichsprüfung ausgeschaltet ist ({$R-}), variieren die *code*-Werte für außerhalb des Bereichs liegende Werte abhängig vom Datentyp von *v*. Wenn *v* den Typ *Longint* oder *Real* hat, ist der Wert von *v* undefiniert und *code* liefert einen Wert ungleich Null zurück. Für alle anderen numerischen Typen liefert *code* einen Nullwert zurück und *v* enthält das Ergebnis des Überlaufs (unter der Annahme, daß der String-Wert im Longint-Bereich liegt).

Deswegen sollten Sie *Val* eine Longint-Variable übergeben und Bereichsüberprüfungen durchführen, bevor Sie eine Zuweisung des gelieferten Werts machen:

```
{$R-}
Val('65535', LongIntVar, code)
if (code  0) or LongIntVar  0) or (LongIntVar  65535)
then
  ...    {Error}
else
  WordVar := LongIntVar;
```

In diesem Beispiel wird *LongIntVar* auf 65535 gesetzt und *code* ist 0. Da 65535 außerhalb des Bereichs der Variable *Word* liegt, wird ein Fehler gemeldet.

Grenzen nachfolgende Leerzeichen müssen entfernt werden.

Querverweis *Str*

Beispiel

```
var  i, code: Integer;
begin
  { Interpretiert den 1. Kommandozeilen-
    Parameter als numerischen Wert }
  Val(ParamStr(1), i, code);
  if code <> 0 then
   Writeln('Kann nicht numerisch interpretiert werden');
  else
   Writeln('numerischer Wert: ', i);
end.
```

WhereX-Prozedur **WinCrt**

Funktion liefert die Y-Koordinate (Spaltenposition) des Cursors (relativ zum momentanen Fenster) zurück.

Deklaration
```
WhereX: Integer
```

Erläuterung der gelieferte Wert basiert auf 1 und entspricht *Cursor.X + 1*.

WhereY-Prozedur **WinCrt**

Funktion liefert die X-Koordinate (Zeilenposition) des Cursors (relativ zum momentanen Fenster) zurück.

Deklaration
```
WhereY: Integer
```

Erläuterung der gelieferte Wert basiert auf 1 und entspricht *Cursor.Y + 1*.

Write-Prozedur

Funktion schreibt Daten in eine Datei.

Deklaration
```
Write(var f, v1 [, v2,...,vn] ) für typisierte Dateien
Write([var f: Text;] v1 [,v2,...,vn])
                                  für Textdateien
```

Erläuterung *Write für typisierte Dateien: f* steht für eine Datei-Variable beliebigen Typs (außer *Text*), jede der angegebenen Variablen *v* hat denselben Typ wie die Komponenten dieser Datei. Der Positionszeiger innerhalb der Datei wird durch jede Schreibaktion um eine Komponente weiterbewegt. Wenn der Positionszeiger vor einer Schreibaktion auf dem Dateiende steht (d.h. *Eof(f)* das Ergebnis *True* hat), dann wird die Datei durch die Schreibaktion vergrößert. Da die momentane Position immer auf die jeweils nächste Komponente gesetzt wird, liefert *Eof(f)* in diesem Fall auch nach der Schreibaktion weiterhin das Ergebnis *True*.

Wenn *Write* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Write für Textdateien: Wenn *f* angegeben ist, muß dieser Parameter für eine Textdatei-Variable stehen ist *f* nicht angegeben, dann schreibt *Write* zur Standard-Ausgabe (*Output*).

Jede Angabe *v* in der Syntaxdefinition von *Write* steht für einen Ausdruck (»OutExpr«), dessen Wert in die Datei geschrieben wird. Jedem Ausdruck kann die Angabe einer Feldbreite folgen, für Ausdrücke mit Real-Typen ist zusätzlich die

Angabe der gewünschten Anzahl von Nachkommastellen möglich. Jeder Ausgabe-Ausdruck muß entweder ein Char-Typ, ein Integer-Typ, ein Real-Typ, ein *String* oder *Boolean sein.*

Die Parameter für *Write* werden folgendermaßen notiert:

```
OutExpr [ : MinBreite [ : DezStellen]]
```

MinBreite und *DezStellen* sind Integer-Ausdrücke. *MinBreite* definiert die minimale Breite des Feldes und muß größer als 0 sein. Es werden mindestens soviele Zeichen ausgeschrieben, wie durch *MinBreite* vorgegeben sind; enthält die Zeichenkette weniger Zeichen, werden entsprechend viele Leerzeichen vorangestellt. *OutExpr* wird durch eine zu niedrige Angabe von *MinBreite* nicht abgeschnitten: es werden immer mindestens so viele Zeichen ausgegeben, wie zur Darstellung von *OutExpr* notwendig sind. Dasselbe gilt, wenn *MinBreite* nicht angegeben ist.

DezStellen definiert die Anzahl der Dezimalstellen in einer Fixkomma-Darstellung eines Real-Typs und kann nur angegeben werden, wenn *OutExpr* einen Real-Typ hat und *MinBreite* ebenfalls definiert ist. *DezStellen* ist ein Integer-Ausdruck und muß einen Wert größer Null haben.

Write mit einem Char-Typen: Ist *MinBreite* nicht angegeben, wird das durch *OutExpr* dargestellte Zeichen in die Datei geschrieben, ansonsten werden *MinBreite*-1 Leerzeichen vorangestellt.

Write mit einem Integer-Typen: Wenn *MinBreite* nicht angegeben ist, wird die dezimale Darstellung von *OutExpr* ohne führende Leerzeichen in die Datei geschrieben. Ist *MinBreite* definiert und dieser Wert größer als die Länge der Ziffernfolge, werden entsprechend viele Leerzeichen vorangestellt.

Write mit einem Real-Typen: Wenn *OutExpr* einen Real-Typ hat, wird der entsprechende Wert in dezimaler Darstellung in die Datei geschrieben. Das Format dieser Darstellung wird über den Wert von *DezStellen* bzw. durch die Abwesenheit dieses Parameters bestimmt.

Wenn *DezStellen* nicht angegeben ist, wird die Fließkomma-Notation benutzt. Ist auch *MinBreite* nicht angegeben, dann benutzt Turbo Pascal den Wert 10 als Standardvorgabe. Angaben für *MinBreite* außerhalb des Bereichs von 10..80 haben dieselbe Wirkung wie eine Angabe des Wertes 10. Das Format eines dezimalen Strings ist:

```
[ - ] <Ziffer> . <Nachkst.> e [ + | - ] <Exponent>
```

Die einzelnen Komponenten dieses Diagramms haben folgende Bedeutungen:

[-]	Nichts oder »-«, entsprechend dem Vorzeichen von *OutExpr*
<Ziffer>	Einzelne Ziffer, nur »0«, wenn auch *OutExpr* 0 ist
.	Dezimalpunkt
<Nachkst.>	Ziffernfolge mit *MinBreite* -9 Ziffern
e	Kleingeschriebenes »e«
[+ | -]	»+« oder »-«, entsprechend dem Vorzeichen des Exponenten
<Exponent>	Exponent mit nachfolgenden Leerzeichen, wenn weniger als 4 Stellen

Wenn *DezStellen* angegeben ist, wird ein String in Fixkomma-Notation geschrieben, der folgendes Format hat:

```
[ <Leerz.> ] [ - ] <Ziffern> [ . <Nachkst.> ]
```

Die einzelnen Komponenten dieses Diagramms haben folgende Bedeutung:

[<Leerz.>]	Leerzeichen zur Angleichung an die Vorgabe von *MinBreite*
[-]	»-«, wenn *OutExpr* negativ ist
<Ziffern>	mindestens eine Ziffer, keine führenden Nullen
.	Dezimalpunkt (wenn *DezStellen* > 0)
<Nachkst.>	Nachkommastellen (wenn DezStellen >0)

Wenn *DezStellen* angegeben und *OutExpr* größer oder gleich 10^ (19 - *DezStellen*) ist, wird die Zahl in Fließkomma-Notation mit 19 signifikanten Ziffern formatiert, wobei entsprechend dem Wert von *MinBreite* Leerzeichen vorangestellt werden.

Write mit einem String-Typen: Ist *MinBreite* nicht angegeben, wird der Stringwert von *OutExpr* ohne führende Leerzeichen in die Datei geschrieben. Wenn *MinBreite* angegeben und dieser Wert größer als die Länge von *OutExpr* ist, wird eine entsprechende Anzahl von Leerzeichen vorangestellt.

Write mit einem gepackten String-Typen: Wenn *OutExpr* den gepackten Stringtyp hat, sind die Auswirkungen dieselben wie beim Schreiben eines Strings, dessen Länge die Anzahl der Elemente im gepackten Stringtyp ist.

Write mit einem *booleschen Wert*: Wenn *OutExpr* den Typ *Boolean* hat, repräsentiert *Write* den Wert durch die Zeichenfolgen *True* oder *False*, die wie Strings entsprechender Länge behandelt werden.

Wenn *Write* fehlerfrei ausgeführt werden konnte, dann liefert ein nachfolgender Aufruf von *IOResult* im Modus {$I-} den Wert 0, ansonsten hat *IOResult* den entsprechenden Fehlercode (<> 0) als Ergebnis.

Grenzen die durch *f* bezeichnete Datei muß offen sein.

Querverweis *Read, Readln, Writeln*

WriteBuf-Funktion **WinCrt**

Funktion schreibt einen Block von Zeichen in ein WinCrt-Fenster.

Deklaration `WriteBuf(Buffer: PChar; Count: Word)`

Ergebnistyp *Word*

Erläuterung *Buffer* zeigt auf das erste Zeichen des Blocks. *Count* enthält die Anzahl der Zeichen, die geschrieben werden sollen. Wenn *Autotracking* auf *True* gesetzt ist, wird der Inhalt des Fensters falls nötig verschoben, um sicherzustellen, daß der Cursor sichtbar bleibt, nachdem die Zeichen geschrieben wurden.

WriteChar-Prozedur **WinCrt**

Funktion schreibt ein einzelnes Zeichen in ein WinCrt-Fenster.

Deklaration `WriteChar(Char: PChar)`

Erläuterung schreibt das Zeichen *Ch* an der aktuellen Cursorposition durch Aufruf von WriteBuf(@Ch,1).

Querverweis *WriteBuf*

Writeln-Prozedur

Funktion schreibt Daten in eine Textdatei und schließt mit einem Zeilenvorschub ab..i.Ausgabe:Writeln-Prozedur;

Deklaration `Writeln([var f: Text;] v1 [,v2,...,vn])`

Erläuterung *Writeln* ist nur auf Textdateien anwendbar und behandelt angegebene Parameter auf dieselbe Art wie *Write*. Der einzige Unterschied besteht darin, daß *Writeln* den Schreibvorgang mit der Ausgabe eines Zeilenvorschubs beendet. *Writeln(f)* ohne weitere Parameter schreibt ein Zeilenende-Zeichen in die durch *f* angegebene Datei; *Writeln* ohne Angabe einer Datei-Variablen entspricht *Writeln(Output)* und erzeugt einen Zeilenvorschub auf dem Bildschirm.

Grenzen die durch *f* bezeichnete Datei muß offen sein und den Typ *Text* haben.

Querverweis *Write*

VI. Anhang

A. Fehlermeldungen und -codes

Turbo Pascal unterscheidet (wie jeder andere Compiler auch) zwischen zwei Arten von Fehlern, die eigentlich recht wenig miteinander zu tun haben:

- *Fehler während der Compilierung* werden durch *syntaktisch* unzulässige Konstrukte wie fehlende oder überzählige Elemente, falschgeschriebene Schlüsselwörter und/oder Variablennamen sowie durch Beschränkungen des Compilers und des Systems (mehr als 64 KByte Code in einem Segment, zuviele Units, mehr als 64 KByte globale Variablen usw.) hervorgerufen. Die dazugehörigen Meldungen werden vom Compiler ausgegeben.

- *Fehler während des Programmlaufs* auch allgemein als *Laufzeitfehler* bezeichnet entstehen beispielsweise beim Versuch, eine Variable über ihren zulässigen Wertebereich hinaus zu erhöhen oder Zeichen zur Tastatur zu schreiben. Da der Compiler keine Annahmen über den Verlauf des Programms macht er kann beispielsweise nicht bestimmen, wie oft eine mit **while** aufgebaute Schleife wiederholt wird treten diese *semantischen* Fehler erst zur Laufzeit des Programms in Erscheinung. Die Codes dieser Fehler werden durch Routinen der Laufzeitbibliothek definiert. Meldungen im Klartext erhalten Sie nur dann, wenn das Programm innerhalb der integrierten Entwicklungsumgebung gestartet wurde.

Fehlermeldungen des Compilers

Wenn der Compiler einen Fehler erkennt, gibt er nicht nur eine entsprechende Meldung aus, sondern auch noch soviele Zusatzinformationen wie möglich in Form von Bezeichnern oder Dateinamen. Ein Beispiel:

```
Error 15:File not found (WINDOW.TPU)
```

Innerhalb der integrierten Entwicklungsumgebung wird die Datei, in der der Fehler erkannt wurde, automatisch in den Editor geladen und der Cursor entsprechend positioniert. Die Kommandozeilen-Version des Compilers gibt eine Fehlermeldung zusammen mit der Zeile des Quelltextes aus, in der der Fehler auftrat. Auf die Fehlerstelle wird durch das Zeichen ^ hingewiesen. Der Punkt, an dem ein Fehler vom Compiler erkannt wird, liegt in verschiedenen

Fällen einige Zeilen hinter der tatsächlichen Fehlerstelle im Quelltext. Ist zum Beispiel der Typ einer Variablen nicht kompatibel mit dem ihr zugewiesenen Wert, kann dieser Fehler erst ermittelt werden, wenn der Ausdruck nach dem Zuweisungsoperator := vollständig ausgewertet wurde.

1 Out of memory
(Platz im Hauptspeicher reicht nicht aus)
Für diesen Fehler gibt es eine Reihe von Gründen und Möglichkeiten zur Behebung:

- Versuchen Sie, den unter Windows zur Verfügung stehenden Arbeitsspeicher zu vergrößern, zum Beispiel dadurch, daß Sie andere, gerade aktive Anwendungen schließen.
- Legen Sie über *Options/Linker/Link buffer* fest, daß der Link-Puffer auf Disk angelegt wird oder verwenden Sie die Option /L in der Kommandozeilenversion.

Der letzte (und komplizierteste) Ausweg: Verteilen Sie Ihr Programm auf mehrere Units und compilieren Sie diese Units separat.

2 Identifier expected
(Bezeichner erwartet)
Der Compiler erwartet an dieser Stelle einen Bezeichner bzw. Ihre Angabe ist aus irgendeinem Grund nicht als Bezeichner akzeptabel. Diesen Fehler erhalten Sie auch beim Versuch, ein reserviertes Wort zu redeklarieren.

3 Unknown identifier
(unbekannter Bezeichner)
Der angegebene Bezeichner ist dem Compiler unbekannt (entweder haben Sie die Deklaration vergessen oder sich irgendwo vertippt).

4 Duplicate identifier
(doppelter Bezeichner)
Der angegebene Bezeichner ist innerhalb des momentanen Blocks bereits vergeben.

5 Syntax error
(Syntaxfehler)
Der Compiler hat ein unzulässiges Zeichen innerhalb des Quelltextes entdeckt. (Möglicher Grund: Ein Steuerzeichen für den Drucker o.ä. außerhalb eines Kommentars oder außerhalb eines mit Hochkommata eingeschlossenen Strings).

6 Error in real constant
(fehlerhafte Angabe einer Real-Konstanten)
Die Syntax von Real-Konstanten ist in Kapitel 1 »Sprachelemente und Konstanten« beschrieben.

7 Error in integer constant
(fehlerhafte Angabe einer Integer-Konstanten)
Ganzzahlige Werte müssen mit einem Dezimalpunkt und einer folgenden Null angegeben werden, wenn sie außerhalb des Wertebereichs für Integertypen liegen (Beispiel: 8844349587.0). Die Syntax von Integer-Konstanten ist in Kapitel 1 »Sprachelemente und Konstanten« beschrieben.

8 String constant exceeds line
(String-Konstante geht über das Zeilenende hinaus)
String-Konstanten müssen grundsätzlich in der Zeile des Quelltextes abgeschlossen werden, in der sie auch begonnen haben. Wahrscheinlich haben Sie ein abschließendes Hochkomma vergessen.

9 Too many nested files
(Include-Verschachtelung zu tief)
Verschachtelungen über den Compiler-Befehl {$I <Dateiname>} sind maximal bis zu einer Tiefe von 15 Ebenen erlaubt.

10 Unexpected end of file
(unerwartetes Quelltext-Ende)
Für diese Fehlermeldung gibt es drei wahrscheinliche Gründe:

- Ihre Quelltext-Datei endet vor dem abschließenden **end.** des Hauptprogramms, Sie haben das Hauptprogramm versehentlich mit einem Semikolon (**end;**) abgeschlossen oder der Text enthält allgemein eine nicht ausgeglichene Anzahl von **begin** und **end**.
- Eine Include-Datei endet inmitten eines Anweisungsteils. (Die Aufteilung eines Prozedur- oder Funktionsblocks auf mehrere Dateien ist nicht möglich).
- Sie haben vergessen, einen Kommentar abzuschließen.

11 Line too long
(Programmzeile hat mehr als 126 Zeichen)

12 Type identifier expected
(Typ-Bezeichner erwartet)

13 Too many open files
(zuviele Dateien auf einmal offen)
Das System reserviert nicht genügend Speicherplatz für gleichzeitig offene Dateien. Prüfen Sie den Parameter FILES in der Datei CONFIG.SYS und erhöhen Sie ihn gegebenenfalls (FILES=20). Danach ist ein Neustart des Computers erforderlich, um die Änderungen wirksam werden zu lassen.

14 Invalid file name
(ungültiger Dateiname)
Der in einem Compiler-Befehl angegebene Dateiname und/oder Suchweg ist ungültig bzw. der Suchweg existiert nicht.

15 File not found
(Datei nicht gefunden)
Die in einem Compiler-Befehl angegebene Datei läßt sich weder im momentan gesetzten Directory noch über die Suchwege (*Options/ Directories*) für den entsprechenden Dateityp finden. Diesen Fehler erhalten Sie auch dann, wenn der Compiler ein mit **uses** angebenes Unit nicht findet.

16 Disk full
(Diskette/Festplatte voll)
Löschen Sie nicht mehr benötigte Dateien (*.BAK).

17 Invalid compiler directive
(Compiler-Befehl ungültig)
Entweder ist der nach {$ angegebene Buchstabe nicht als Compiler-Befehl definiert, ein angegebener Parameter ungültig oder Sie haben einen Befehl im Rumpf des Programms angegeben, der globale Wirkung hat und deshalb nur vor der ersten Prozedur- oder Funktionsdeklaration gegeben werden darf.

18 Too many files
(zuviele Dateien innerhalb eines Projekts)
Die interne Dateinamen-Tabelle des Compilers ist voll. Nehmen Sie Include-Dateien direkt in Ihre Quelltexte auf, fassen Sie sämtliche Dateien in einem Verzeichnis zusammen, das als Standard gesetzt wird, und verwenden Sie kürzere Dateinamen.

19 Undefined type in pointer definition
(Ziel-Typ der Zeigerdefinition ist nicht definiert)
Der bei der Definition eines Zeigers angegebene Datentyp ist dem Compiler nicht bekannt.

20 Variable identifier expected
(Variablen-Bezeichner erwartet)

21 Error in type
(unzulässiges Symbol/Zeichen in einer Typ-Definition)

22 Structure too large
(Datenstruktur ist größer als 65520 Bytes)

23 Set base type out of range
(Mengen-Grundtyp außerhalb des möglichen Wertebereichs)
Der Grundtyp einer Menge muß ein Teilbereich sein, dessen untere Grenze größer/gleich 0 und dessen obere Grenze kleiner/gleich 255 ist, oder ein Aufzählungstyp mit maximal 256 verschiedenen Werten.

24 File components may not be files or objects
[Dateikomponenten dürfen keine Dateien oder Objekte sein.]
Als zusammengesetzten Datentyp einer Datei darf man keinen Objekt- oder

Dateityp angeben. Es darf auch kein strukturierter Datentyp mit einer Objekt- oder Dateikomponente angegeben werden.

25 Invalid string length
(Stringlänge außerhalb des Bereichs 1..255)

26 Type mismatch
(Typen nicht miteinander vereinbar)
Dieser Fehler kann einen der folgenden Gründe haben:

- Bei Zuweisungen: Der Typ der Variablen und der Ergebnistyp des Ausdrucks sind nicht implizit kompatibel (d.h. der Compiler nimmt eine entsprechende Typ-Umwandlung nicht selbständig vor).
- Bei Prozedur- und Funktionsaufrufen: Formaler und aktueller Parameter sind nicht implizit kompatibel.
- Bei Indizierungen: Der Ergebnistyp des Ausdrucks und der Indextyp des Arrays sind nicht kompatibel.
- Bei Ausdrücken: Die Kombination von Operanden und Operatoren ist nicht miteinander verträglich (Vergleich von *Integer* und *Char* etc).

27 Invalid subrange base type
(Grundtyp des Teilbereichs ist nicht ordinal)

28 Lower bound greater than upper bound
(Untergrenze eines Teilbereichs > Obergrenze)

29 Ordinal type expected
(Ordinaltyp erwartet)
Real-Typen, Strings, strukturierte Typen und Zeiger sind an dieser Stelle nicht verwendbar.

30 Integer constant expected
(Integer-Konstante erwartet)

31 Constant expected
(Konstante erwartet)

32 Integer or real constant expected
(Integer- oder Real-Konstante erwartet)

33 Pointer type identifier expected
(Zeiger-Typ-Bezeichner erwartet)
Der Bezeichner repräsentiert keinen Zeiger-Typ

34 Invalid function result type
(Ergebnistyp der Funktion nicht zulässig)

Gültige Typen von Funktionsergebnissen sind alle einfachen Typen, Strings und Zeiger.

35 Label identifier expected
(Label-Bezeichner erwartet)

36 BEGIN expected
(**begin** erwartet)

37 END expected
(**end** erwartet)

38 Integer expression expected
(Ergebnistyp des Ausdrucks muß *Integer* sein)

39 Ordinal expression expected
(Ergebnistyp des Ausdrucks muß ordinal sein)

40 Boolean expression expected
(Ergebnistyp des Ausdrucks muß *Boolean* sein)

41 Operand types do not match operator
(Operandentypen passen nicht zum Operator)
Der Operator kann auf Operanden dieses Typs nicht angewendet werden (z.B. 'A' **div** '2').

42 Error in expression
(Fehler innerhalb des Ausdrucks)
Der Compiler hat zwar verstanden, daß es sich um einen Ausdruck handelt, kann diesen Ausdruck aber nicht auswerten (z.B. wegen eines vergessenen Operators).

43 Illegal assignment
(Zuordnung nicht erlaubt)
Die wahrscheinlichsten Gründe für diese Fehlermeldung sind:

- Der Versuch, Datei-Variablen oder einer untypisierten Variablen direkt einen Wert zuzuordnen.
- Der Versuch, einer Funktion außerhalb ihres Blocks einen Wert zuzuordnen.

44 Field identifier expected
(Feld-Bezeichner erwartet)
Der angegebene Bezeichner steht nicht für ein Feld der angegebenen Record-Variablen.

45 Object file too large
(Linker: .OBJ-Datei ist größer als 64 KByte)

46 Undefined external
(Linker: **external**-Bezeichner ist nirgendwo definiert)
Ein als **external** deklarierter Bezeichner hat sich in keiner der beteiligten Dateien gefunden. Prüfen Sie, ob sämtliche benötigten Object-Dateien auch über den Compiler-Befehl {$L <Dateiname>} angegeben sind, dann Ihre PUBLIC-Deklarationen innerhalb der Assembler-Quelltexte und schließlich auf eventuelle Tippfehler.

47 Invalid object file record
(Linker: .OBJ-Dateiformat unleserlich)
Der Linker kann eine via {$L <Dateiname>} angegebene Datei nicht fehlerfrei einbinden.

48 Code segment too large
(Codesegment-Grenzen überschritten)
Die Compilierung eines Programms oder einer Unit erzeugt mehr als 65520 Bytes Code. Erstellen Sie eine zusätzliche Unit, in das Sie einige Prozeduren und/oder Funktionen »auslagern«, und nehmen Sie diese Unit mit **uses** in das ursprüngliche Programm (oder Unit) auf.

49 Data segment too large
(Datensegment-Grenzen überschritten)
Ihr Programm und sämtliche aufgenommenen Units deklarieren zusammen mehr als 64 KByte globaler Daten. Es bleibt Ihnen nichts anderes übrig, als zumindest die größeren Datenstrukturen über Zeiger zu deklarieren und sie über Aufrufe von *New* dynamisch zu erzeugen (d.h. Sie müssen wahrscheinlich größere Teile Ihres Programms umschreiben).

50 DO expected
(**do** erwartet)

51 Invalid PUBLIC definition
(Linker: PUBLIC-Definition inkorrekt)
Diese Fehlermeldung können Sie nur beim Einbinden von Maschinenprogrammen erhalten. Mögliche Gründe sind:

- Ein Bezeichner ist über PUBLIC als global verfügbar definiert worden, der Pascal-Quelltext enthält aber keine entsprechende **external**-Deklaration.
- Ein und derselbe Bezeichner ist in mehreren Object-Dateien als PUBLIC definiert worden.
- Ein als PUBLIC definierter Bezeichner befindet sich außerhalb des CODE-Segments in der .OBJ-Datei.

52 Invalid EXTRN definition
(Linker: EXTRN-Definition inkorrekt)
Diese Fehlermeldung können Sie nur beim Einbinden von Maschinenprogrammen erhalten. Mögliche Gründe sind:

- Ein Bezeichner ist innerhalb des Maschinenprogramms als EXTRN definiert, findet sich aber in keiner Pascal-Datei als global verfügbar (d.h. als eigenständige Prozedur/Funktion innerhalb des Programms oder als Deklaration innerhalb eines Interface-Teils).

- Der Bezeichner ist im Pascal-Programm als **absolute** deklariert.

- Der Bezeichner steht innerhalb des Pascal-Programms für eine als **inline** deklarierte Routine.

53 Too many EXTRN definitions
(Linker: zuviele EXTRN-Definitionen)
Maximal sind 256 EXTRN-Definitionen pro Object-Datei möglich.

54 OF expected
(**of** erwartet)

55 INTERFACE expected
(**interface** erwartet)

56 Invalid relocatable reference
(Zugriff auf relozierbares Symbol inkorrekt)
Diese Fehlermeldung können Sie nur beim Einbinden von Maschinenprogrammen erhalten. Mögliche Gründe sind:

- Die .OBJ-Datei definiert relozierbare Symbole außerhalb des Segments CODE.

- Die .OBJ-Datei enthält Zugriffe auf einzelne Bytes relozierbarer Symbole beispielsweise die Operatoren HIGH und LOW oder die DB-Direktiven (define byte).

- Ein Operand innerhalb der .OBJ-Datei bezieht sich auf ein relozierbares Symbol, das weder innerhalb des Segments CODE noch innerhalb des Segments DATA definiert ist.

- Ein Operand innerhalb der .OBJ-Datei bezieht sich auf eine als EXTRN deklarierte Prozedur oder Funktion über eine relative Adreßangabe, wie beispielsweise der Befehl *CALL PascalSort+8*.

57 THEN expected
(**then** erwartet)

58 TO or DOWNTO expected
(**to** oder **downto** erwartet)

59 Undefined forward
(**forward**-Deklaration: Definition fehlt)
Diese Fehlermeldung kann zwei verschiedene Gründe haben:

- Die Prozedur oder Funktion wurde im Interface-Teil einer Unit deklariert, aber im Implementations-Teil nicht definiert.

■ Die Prozedur oder Funktion wurde in einem Programm als **forward** deklariert, der Compiler hat aber die tatsächliche Definition nicht gefunden.

61 Invalid typecast
(Typ-Umwandlung nicht möglich)
Entweder stimmen die Größen von Original- und Ziel-Typ nicht überein oder Sie versuchen die Typ-Umwandlung eines Ausdrucks an einer Stelle, an der nur Variablen-Zugriffe stehen können.

62 Division by zero
(Division durch Null)
Der Compiler berechnet konstante Ausdrücke bereits während der Übersetzung und hat dabei offensichtlich eine Division durch Null vornehmen müssen.

63 Invalid file type
(Operation mit diesem Dateityp nicht erlaubt)
Die Operation kann auf eine Dateivariable des angegebenen Typs nicht angewendet werden (wie *Seek* auf eine Textdatei oder *Readln* auf eine typisierte Datei).

64 Cannot Read or Write Variables of this type
(Variablen dieses Typs können nicht gelesen/geschrieben werden)
Die Ausgabe von strukturierten Typen (außer Strings) und Zeigerwerten ist mit *Write* und *Writeln* nicht möglich, *Read* und *Readln* haben dieselben Grenzen. Boolsche Werte können mit *Write* und *Writeln* ausgegeben, nicht aber mit *Read* oder *Readln* eingelesen werden.

65 Pointer variable expected
(Zeiger-Variable erwartet)

66 String variable expected
(String-Variable erwartet)

67 String expression expected
(String-Ausdruck erwartet)

68 Circular unit reference
(überkreuzender Bezug zweier Units)
Sie haben zwei Units definiert, die sich gegenseitig im Interface-Teil voraussetzen, also beispielsweise:

```
unit U1;          unit U2;
uses U2;          uses U1;
```

Derartige Bezüge sind in Turbo Pascal nur über den Implementations-Teil erlaubt.

69 Unit name mismatch
(Unitname und Unit-Dateiname stimmen nicht überein)
Die Unit hat einen anderen Namen als die angegebene .TPU-Datei. (Der .TPU-

Dateiname und der als **unit** deklarierte Bezeichner müssen grundsätzlich gleich sein).

70 Unit version mismatch
(Unit-Versionen stimmen nicht überein)
Eine oder mehrere Units, die von dieser Unit vorausgesetzt werden, sind seit der letzten Compilierung verändert worden. Verwenden Sie *Compile/Build* oder *Compile/Make* der integrierten Entwicklungsumgebung bzw. die Kommandozeilen-Parameter /M oder /B, damit Turbo Pascal diese Unit compiliert und damit auf den neusten Stand bringt.

72 Unit file format error
(Linker: Unit-Dateiformat ungültig)
Der Linker kann den Inhalt der angegebenen Datei nicht als Unit interpretieren.

73 IMPLEMENTATION expected
(**implementation** erwartet)

74 Constant and case types do not match
(**case**-Konstante und Selektor haben verschiedene Datentypen)

75 Record variable expected
(Record-Variable erwartet)

76 Constant out of range
(Konstante außerhalb des zulässigen Wertebereichs)
Für diese Fehlermeldung kann es mehrere Gründe geben:

- Indizierung eines Arrays mit einer Konstanten, die außerhalb der Arraygrenzen liegt.
- Zuweisung eines (konstanten) Wertes an eine Variable, der außerhalb des Wertebereichs der Variablen liegt.
- Übergabe eines (konstanten) Parameters an eine Prozedur oder Funktion, dessen Wert außerhalb des Wertebereichs des entsprechenden formalen Parameters liegt.

77 File variable expected
(Datei-Variable erwartet)

78 Pointer expression expected
(Zeiger-Ausdruck erwartet)

79 Integer or real expression expected
(Integer- oder Real-Ausdruck erwartet)

80 Label not within current block
(**goto**-Sprungziel muß innerhalb des momentanen Blocks liegen)

81 Label already defined
(Label wurde bereits zur Markierung einer Anweisung verwendet)

82 Undefined label in preceding statement part
(vorangehender Anweisungsteil läßt Label undefiniert)
Innerhalb des vorangehenden Blocks wurde ein Label deklariert und auch als Sprungziel von **goto** verwendet; dem Compiler fehlt aber eine Definition, d.h. er kann nicht festlegen, wohin mit dem **goto** gesprungen werden soll.

83 Invalid @ argument
(unzulässiges Argument für **@**)
Als Argument für den Adreß-Operator **@** können nur Variablenbezüge sowie Prozedur- und Funktionsbezeichner verwendet werden. **inline**-Prozeduren und Funktionen sind ebenfalls keine zulässigen Argumente für **@**.

84 UNIT expected
(**unit** erwartet)

85 ";" expected
(";" erwartet)

86 ":" expected
(":" erwartet)

87 "," expected
("," erwartet)

88 "(" expected
("(" erwartet)

89 ")" expected
(")" erwartet)

90 "=" expected
("=" erwartet)

91 ":=" expected
(":=" erwartet)

92 "[" or "(." expected
("[" oder "(." erwartet)

93 "]" or ".)" expected
("]" oder ".)" erwartet)

94 "." expected
("." erwartet)

95 ".." expected
(".." erwartet)

96 Too many variables
(zuviele Variablen)
Diese Fehlermeldung erhalten Sie, wenn entweder

- die Größe der globalen Variablen eines Programms und darin aufgenommener Units 64 KByte überschreitet, oder
- innerhalb einer Prozedur oder Funktion mehr als 64 KByte lokale Variablen deklariert sind.

97 Invalid FOR control variable
(Variable nicht als Laufvariable einer **for**-Schleife verwendbar)
Als Laufvariable einer **for**-Schleife können nur einfache ordinale Variablen verwendet werden, die entweder global zum Programm oder lokal zum momentanen Block deklariert sein müssen.

98 Integer variable expected
(Integer-Variable erwartet)

99 File and procedure types are not allowed here
[Datei- und Prozedurtypen sind hier nicht erlaubt.]

100 String length mismatch
(Stringlänge und Arraygröße stimmen nicht überein)
Die (absolute) Länge des Strings entspricht nicht der Anzahl der Elemente des angegebenen *Char*-Arrays.

101 Invalid ordering of fields
(Reihenfolge der Felder stimmt nicht überein)
Die Reihenfolge der Felder muß bei einer Record-Konstanten dieselbe sein wie bei der Definition des Recordtyps.

102 String constant expected
(String-Konstante erwartet)

103 Integer or real variable expected
(Integer- oder Real-Variable erwartet)

104 Ordinal variable expected
(Variable ordinalen Typs erwartet)

105 INLINE error
(**inline**: "<"-Operator nicht auf Variablen anwendbar)

106 Character expression expected
(Ausdruck des Ergebnistyps *Char* erwartet)

112 CASE constant out of range
(**case**-Konstante außerhalb des zulässigen Bereichs)
Die Werte der **case**-Konstanten müssen im Bereich von -32768..32767 liegen, solange mit einem *Integer* als Selektor gearbeitet wird.

113 Error in statement
(Syntaxfehler in Befehl/Anweisung)
Mit dem von Ihnen angegebenen Symbol darf eine Anweisung nicht beginnen.

114 Cannot call an interrupt procedure
(direkter Aufruf von **interrupt**-Prozeduren ist nicht erlaubt)

116 Must be in 8087 mode to compile this
(Datentyp/Konstrukt nur im Modus {$N+} möglich)
Operationen mit den Variablentypen *Single*, *Double*, *Extended* und *Comp* setzen voraus, daß der Modus {$N+} eingeschaltet ist.

117 Target address not found
(angegebene Adresse ist nicht Teil des Programms)
Die Suche nach einem Laufzeitfehler (mit *Search/Find Error* bzw. dem Kommandozeilen-Parameter /F) konnte an der angegebenen Adresse keinen Befehl des Programms finden.

118 Include files are not allowed here
(Include-Befehl an dieser Stelle nicht erlaubt)
Der Compiler-Befehl {$I <Dateiname>} kann nicht inmitten eines Blocks angegeben werden.

120 NIL expected
(**nil** erwartet)

121 Invalid qualifier
(ungültige Qualifizierung)
Diese Fehlermeldung erhalten Sie bei dem Versuch,

- eine Variable zu indizieren, die kein Array ist;
- auf Felder einer Variablen zuzugreifen, die kein Record ist;
- über eine Variable *X*, die keinen Zeigertyp hat, auf eine andere Variable zuzugreifen (also mit *X*^).

122 Invalid variable reference
(ungültiger Variablenbezug)
Das angegebene Konstrukt folgt den Regeln eines Variablenbezugs, ergibt aber letztendlich nicht die Speicheradresse einer Variablen. Aller Wahrscheinlichkeit nach haben Sie eine Zeigerfunktion verwendet und vergessen, das Ergebnis dieser Funktion (mit ^) in eine Adresse umzuwandeln.

123 Too many symbols
(Symbole belegen mehr als 64 KByte Speicherplatz)
Das Programm oder die Unit deklariert soviele Symbole, daß 64 KByte Speicherplatz dafür nicht ausreichen. Abgesehen vom »Auslagern« einiger Prozeduren und Funktionen in eine separate Unit haben Sie die Möglichkeit, die Erzeugung von Symbolen mit dem Compiler-Befehl {$D-} zu unterdrücken allerdings ist dann eine Zuordnung von Laufzeitfehlern in diesem Modul nicht mehr möglich.

124 Statement part too large
(Anweisungsteil zu groß)
Turbo Pascal begrenzt den Anweisungsteil einzelner Prozeduren, Funktionen und des Hauptprogramms auf *jeweils* rund 24 KByte Code. Falls Sie diese Fehlermeldung erhalten, müssen Sie einige Befehlsfolgen in zusätzliche Prozeduren oder Funktionen verbannen. (Programme, bei denen ein durchgehender Anweisungsteil diese Grenze überschreitet, haben ein gründliches Aufräumen sowieso dringendst nötig).

126 Files must be var parameters
(Datei-Variablen müssen **var**-Parameter sein)
In den formalen Parameterlisten von Prozeduren und Funktionen müssen Variablen des Typs **File** grundsätzlich als **var**-Parameter deklariert sein, weil sie immer als Adresse übergeben werden.

127 Too many conditional symbols
(zuviele Symbole für die bedingte Compilierung)
Die Tabelle, in die der Compiler Symbole für die bedingte Compilierung einträgt, ist voll. Versuchen Sie, kürzere Symbolnamen zu verwenden und Symbole, die im weiteren Verlauf der Compilierung nicht mehr gebraucht werden, mit {$UNDEF} wieder zu löschen.

128 Misplaced conditional directive
({$ELSE} oder {$ENDIF} ohne vorangehendes {$IF...})
Der Compiler hat ein {$ELSE} oder {$ENDIF} gefunden, ohne daß zuvor eine Bedingung mit {$IFDEF}, {$IFNDEF} oder {$IFOPT} eingeleitet worden wäre.

129 ENDIF directive missing
({$ENDIF} fehlt}
Innerhalb der Quelltext-Datei wurde eine Bedingung mit {$IF...} begonnen, die am Ende der Datei noch nicht mit {$ENDIF} abgeschlossen ist. (Zu jedem {$IF...} gehört ein {$ENDIF}-Befehl innerhalb derselben Datei eine bedingte Compilierung über die Grenzen von Include-Dateien hinweg ist nicht möglich).

130 Error in initial conditional defines
(Direkt definierte bedingte Symbole enthalten Fehler)
Die mit *Options/Compiler/Conditional defines* bzw. dem Kommandozeilen-Parameter /D vor dem Start der Compilierung definierten Symbole sind aus irgendeinem Grund ungültig.

131 Header does not match previous definition
(Prozedur-/Funktions-Definition ungleich Deklaration)
Für diese Fehlermeldung gibt es zwei mögliche Gründe:

- Eine Funktion oder Prozedur führt im Interface-Teil eines Units eine andere Parameterliste als im Implementations-Teil auf.
- Eine Funktion oder Prozedur führt bei der Deklaration als **forward** eine andere Parameterliste auf als bei ihrer Definition.

132 Critical disk error
(Diskettenfehler während der Compilierung)
Mögliche Gründe sind eine offene Laufwerksklappe, schadhafte Disketten usw.

133 Cannot evaluate this expression
(Dieser Ausdruck läßt sich in diesem Kontext nicht auswerten)
Sie haben in einem Konstanten-Ausdruck eine Funktion der Laufzeitbibliothek verwendet, die der Compiler nicht für Konstanten zuläßt (z.B. *Sin*). Eine Beschreibung der Syntax und der verwendbaren Standardfunktionen finden Sie in Kapitel 5 »Typisierte Konstanten«.

136 Invalid indirect reference
(Unzulässiger indirekter Bezug)
Sie haben versucht, auf einen Bezeichner zuzugreifen, dessen Adresse dem Compiler unbekannt ist (wie z.B. bei einer Deklaration der Form **absolute** *xyz*, wobei das momentane Modul keinen Bezeichner *xyz* definiert). Ein weiterer Grund für diese Meldung sind Bezüge auf nicht definierte Variablen durch **inline**-Befehle.

137 Structured variables are not allowed here
(Strukturierte Variablen sind an dieser Stelle nicht erlaubt)
Mit dieser Meldung reagiert der Compiler auf Operatoren und Operationen, die sich nicht mit strukturierten Variablen durchführen lassen (wie z.B. den Versuch, zwei Records miteinander zu multiplizieren oder ein Array mit *Writeln* auszugeben).

140 Invalid floating point operation
(Fließkomma-Rechenfehler)
Mit dieser Meldung reagiert der Compiler, wenn die Bewertung einer Konstanten mit einer Division durch Null oder einem Überlauf endet.

142 Procedure or function variable expected
(Prozedur- oder Funktions-Variable erwartet)
Der Compiler erwartet an dieser Stelle den Bezeichner einer Prozedur bzw. Funktion (beispielsweise, wenn auf der linken Seite einer Zuweisung eine Prozedur-Variable steht).

143 Invalid procedure or function reference
(Bezug auf Prozedur/Funktion in dieser Weise nicht zulässig)
Diese Meldung kann zwei recht verschiedene Gründe haben:

- der Name einer Prozedur erscheint innerhalb eines Ausdrucks;
- der Compiler moniert die Zuweisung einer Routine an eine Prozedur-Variable: die Routine wurde entweder nicht mit {$F+} übersetzt oder ist als **inline** oder **interrupt** deklariert.

146 File access denied
[Zugriff auf Datei verweigert.]

147 Object type expected.
(Objekttyp erwartet.)
Die Variablenbezeichnung steht nicht für einen Objekttyp.

148 Local object types are not allowed.
(Lokale Objekttypen sind nicht zulässig.)
Objekttypen dürfen nur im äußersten Bezugsrahmen eines Programms oder einer Unit definiert werden. Die Definition eines Objekttyps innerhalb von Prozeduren und Funktionen ist nicht zulässig.

149 VIRTUAL expected.
(VIRTUAL erwartet.)
Das reservierte Wort **virtual** fehlt.

150 Method identifier expected.
(Bezeichner für Methode erwartet.)
Der angegebene Bezeichner ist keine Methode.

151 Virtual constructors are not allowed.
(Virtuelle Konstruktoren sind nicht zugelassen.)
Die Methode für einen Konstruktor muß statisch sein.

152 Constructor identifier expected.
(Bezeichner für einen Konstruktor erwartet.)
Der angegebene Bezeichner ist kein Konstruktor.

153 Destructor identifier expected.
(Bezeichner für einen Destruktor erwartet.)
Der angegebene Bezeichner ist kein Destruktor.

154 Fail only allowed within constructors.
(Fail darf nur innerhalb von Kontruktoren verwendet werden.)
Die Standardprozedur **Fail** darf nur innerhalb eines Konstruktors verwendet werden.

155 Invalid combination of opcode and operands.
(Ungültige Kombination von Opcodes und Operanden)
Der Assembler-Opcode akzepitert diese Operandenkombination nicht. Mögliche Ursachen dafür sind:

- Es wurden zu viele bzw. zu wenige Operanden für einen Assembler-Opcode angegeben, z.B. **INC AX,BX** oder **MOV AX**
- Die Anzahl der Operanden ist zwar korrekt, ihr Typ und/oder ihre Reihenfolge ist jedoch falsch, z.B. **DEC 1**, **MOV AX,CL** oder **MOV 1,AX**

156 Memory reference expected.
(Speicherreferenz erwartet)
Der Assembler-Operrand stellt keine hier benötigte Speicherreferenz dar. In der Regel haben Sie lediglich vergessen, eckige Klammern um einen Index-Register-Operanden zu setzen, z.B. **MOV AX,BX+SI** anstelle von **MOV AX,[BX+SI]**.

157 Cannot add or substract relocatable symbols.
(Kann verschiebbares Symbol nicht addieren oder subtrahieren)
Die einzige arithmetische Operation, die an einem verschiebbaren Symbol in einem Assembler-Operanden vorgenommen werden kann, ist die Addition oder Subtraktion von Konstanten. Variablen, Prozeduren, Funktionen und Labels sind verschiebbare Symbole.
Ein Beispiel: *Var* sei eine Variable und *Const* eine Konstante, dann sind **MOV AX, Const+Const** sowie **MOV AX, Var+Const** gültige Ausdrücke, der Ausdruck **MOV AX, Var+Var** hingegen ist ungültig.

158 Invalid register combination.
(Ungültige Kombination von Registern)
Gültige Kombinationen von Index-Registern sind: `BX` , **[BP]**, **[SI]**, **[DI]**, `BX+SI` , **[BX+DI]**, `BP+SI` und `BP+DI` . Andere Kombinationen von Index-Registern, wie `BP+BX` und `SI+DI` sind nicht erlaubt.
Hinweis: Lokale Variablen (Variablen, die in Prozeduren bzw. Funktionen deklariert sind) werden immer dem Stack zugeordnet, der Zugriff auf sie erfolgt über das BP-Register

159 286/287 Instructions are not enabled.
(286/287 Anweisungen sind nicht zugelassen)
Aktivieren Sie mit dem Compiler-Schalter {$G+} die Erzeugung der 286/287-Opcodes. Bedenken Sie dabei jedoch, daß ein auf diese Weise compiliertes Programm auf Rechnern mit 8086- oder 8088-Prozessoeren nicht mehr lauffähig ist.

160 Invalid symbol reference.
(Ungültige Symbol-Referenz)
Auf dieses Symbol kann in einem Assembler-Operanden nicht zugegriffen werden. Mögliche Gründe dafür sind:

- Sie versuchen in einem Assembler-Operanden auf eine Standard-Prozedur

bzw. -Funktion oder auf die speziellen Arrays *Mem*, *MemW*, *MemL*, *Port* oder *PortW* zuzugreifen.

- Sie versuchen in einem Assembler-Operanden auf eine String-, Gleitkomma- oder Mengen-Konstante zuzugreifen.
- Sie versuchen in einem Assembler-Operanden auf eine **Inline**-Prozedur oder -Funktion zuzugreifen.
- Sie versuchen außerhalb einer Funktion auf das *@Result*-Symbol zuzugreifen.
- Sie versuchen eine short **JMP**-Anweisung zu erzeugen, die nicht auf ein Label verzweigt.

161 Code generation error.
(Fehler bei Code-Erzeugeung)
Der vorangehende Anweisungsteil enthält eine LOOPNE-, LOOPE-, LOOP- oder JCXZ-Anweisung, deren Ziel-Label nicht erreichbar ist.

162 ASM expected.
(ASM erwarted)

163 Duplicate dynamic method index
(Doppelter Index für dynamische Methode)
Der Index für eine dynamische Methode wurde bereits für eine andere Methode vergeben. Vielleicht wollten Sie eine dynamische Methode überschreiben, haben aber den Bezeichner falsch angegeben, und damit unabsichtlich eine neue Methode definiert.

164 Duplicate resource identifier.
(Doppelter Ressourcen-Bezeichner)
Der Bezeichner oder die ID-Nummer einer Ressource wurde bereits für eine andere Ressource verwendet.

165 Duplicate or invalid export index
(Doppelter oder ungültiger **export**-Index)
Der in der **index**-Anweisung angegebene Wert ist entweder nicht im zulässigen Bereich von 1..32767 oder wurde bereits für eine andere zu exportierende Routine verwendet.

166 Procedure or function identifier expected
(Bezeichner einer Prozedur oder Funktion erwartet)
Über die **exports**-Anweisung können nur Prozeduren und Funktionen exportiert werden.

167 Cannot export this symbol
(Dieser Bezeichner kann nicht exportiert werden)
Eine Prozedur oder Funktion kann nur dann exportiert werden, wenn sie mit **export** deklariert wurde.

168 Duplicate export name
(Doppelter Bezeichner in der **name**-Anweisung)
Der in der **name**-Anweisung angegebene Name wurde bereits für eine andere zu exportierende Routine verwendet.

Laufzeitfehler

Turbo Pascal unterscheidet bei Fehlern, die während der Ausführung eines Programms auftreten, zwei Klassen:

Ein-/Ausgabefehler (»I/O Errors») haben Codes im Bereich von 1..199 und können über den Compiler-Befehl {$I-} abgefangen werden.

Fehler mit sofortigem Abbruch (»Fatal Errors«) haben Codes im Bereich von 200..255 und einen unbedingten Abbruch zur Folge; programmgesteuerte Korrekturmaßnahmen sind hier nicht möglich.

Beim Abbruch durch einen Laufzeitfehler erhalten Sie (unabhängig von der Fehlerklasse) eine Meldung des Formats

```
Runtime error nnn at xxxx:yyyy
```

nnn steht dabei für den Fehlercode, *xxxx:yyyy* für die Adresse des Befehls, der den Fehler ausgelöst hat (*xxxx* ist die Segmentadresse, *yyyy* der Offset-Anteil).

Fehlercodes von DOS

1 Invalid function number
(Ungültige Funktionsnummer)
Sie haben eine nicht existierende DOS-Funktion aufgerufen.

2 File not found
(Datei nicht gefunden)
Ausgelöst von *Reset*, *Append*, *Rename* und *Erase*, wenn der der Datei-Variablen zugeordnete Name keine existierende Datei bezeichnet bzw. diese Datei im aktuellen Verzeichnis nicht zu finden ist.

3 Path not found
(Suchweg nicht gefunden)

- Ausgelöst von *Reset*, *Rewrite*, *Append*, *Rename* und *Erase*, wenn der der Datei-Variablen zugeordnete Name einen ungültigen oder nicht existierenden Suchweg angibt.
- Ausgelöst von *ChDir*, *MkDir* und *RmDir*, wenn der angegebene Suchweg ungültig ist oder ein nicht existierendes Verzeichnis bezeichnet.

4 Too many open files
(Maximalzahl an Dateien bereits offen)

Ausgelöst von *Reset*, *Rewrite* und *Append*, wenn das System die Eröffnung einer weiteren Datei aufgrund fehlender Dateipuffer verweigert. Maximal sind 15 Diskettendateien pro Prozeß erlaubt. Falls Sie diesen Fehler früher erhalten, sollten Sie die Datei CONFIG.SYS prüfen und gegebenenfalls um den Befehl FILES=20 erweitern.
Hinweis: Damit Veränderungen von CONFIG.SYS wirksam werden, müssen Sie den Computer neu starten.

5 File access denied
(Dateizugriff verweigert)

- Ausgelöst von *Reset* und *Append*, wenn das Feld *FileMode* der Datei-Variablen Schreib- und Lesezugriffe erlaubt ist und die Datei-Variable einer gegen Überschreiben gesperrten Datei oder einem (Sub-)Directory zugeordnet ist.

- Ausgelöst von *Rewrite*, wenn die Datei-Variable einem (Unter-)Verzeichnis zugeordnet ist oder das entsprechende Verzeichnis bei der Erzeugung einer neuen Datei keinen Platz für den Dateieintrag bietet.

- Ausgelöst von *Rename*, wenn der der Datei-Variablen zugeordnete Name ein (Unter-)Verzeichnis bezeichnet oder bereits eine Datei mit dem neuen Namen existiert.

- Ausgelöst von *Erase*, wenn der der Datei-Variablen zugeordnete Name ein (Unter-)Verzeichnis oder eine gegen Schreibzugriffe gesperrte Datei bezeichnet.

- Ausgelöst von *MkDir*, wenn bereits eine »normale« Datei des angegebenen Namens im Verzeichnis existiert, in dem das Unterverzeichnis eingetragen werden soll, das Verzeichnis keinen Platz für einen weiteren Eintrag bietet oder der angegebene Suchweg ein Peripheriegerät bezeichnet.

- Ausgelöst von *RmDir*, wenn das angegebene Verzeichnis nicht leer ist (d.h. noch Dateieinträge enthält), oder wenn es sich beim angegebenen Suchweg um eine »normale« Datei oder um das Stammverzeichnis des Datenträgers handelt.

- Ausgelöst von *Read* oder *BlockRead* bei einer typisierten bzw. untypisierten Datei, wenn diese Datei nicht für Leseoperationen geöffnet wurde.

- Ausgelöst von *Write* oder *BlockWrite* bei einer typisierten bzw. untypisierten Datei, wenn diese Datei nicht für Schreiboperationen geöffnet wurde.

6 Invalid file handle
(Handle nicht definiert/ungültig)
Dieser Fehler sollte theoretisch nicht vorkommen (solange mit Pascal-Prozeduren gearbeitet wird) falls doch, ist die Datei-Variable aus irgendeinem Grund überschrieben bzw. zerstört worden.

12 Invalid file access mode
(ungültiger Dateimodus)

Ausgelöst von *Reset* und *Append* bei der Anwendung auf typisierte und untypisierte Dateien, wenn der angegebene Zugriff nicht möglich ist, weil das Feld *FileMode* einen ungültigen Wert hat.

15 Invalid drive number
(Laufwerksnummer unzulässig)
Ausgelöst von *GetDir*, wenn die angegebene Laufwerksnummer außerhalb der zulässigen Grenzen liegt.

16 Cannot remove current directory
(als Standard gesetztes Verzeichnis kann nicht gelöscht werden)
Ausgelöst von *RmDir* beim Versuch, das momentan gesetzte Verzeichnis zu löschen.

17 Cannot rename across drives
(Rename kann nicht kopieren)
Ausgelöst von *Rename*, wenn für den alten und den neuen Dateinamen zwei verschiedene Laufwerke angegeben werden.

Ein-/Ausgabefehler

Normalerweise codiert der Compiler nach jedem Ein-/Ausgabebefehl den Aufruf einer Prüfroutine, die im Falle eines Fehlers das Programm abbricht. Ein Programmteil, dem der Compiler-Befehl {$I-} vorangestellt ist, wird dagegen auch im Fehlerfall fortgesetzt hier muß der Programmierer durch einen Aufruf der Funktion *IOResult* explizit prüfen, ob die Operation fehlerfrei verlaufen ist, wobei das Ergebnis 0 für »fehlerfreie Ausführung« steht

100 Disk read error
(Fehler beim Lesen von der Diskette)
Ausgelöst von *Read* beim Versuch, über das Ende einer typisierten Datei hinaus weitere Daten zu lesen.

101 Disk write error
(Fehler beim Schreiben auf die Diskette)
Ausgelöst von *Close*, *Write*, *Writeln*, *Flush* und *Page*, wenn die Diskette voll ist.

102 File not assigned
(Datei-Variable ist keiner Datei zugeordnet)
Ausgelöst von *Reset*, *Rewrite*, *Append*, *Rename* und *Erase*, wenn die Datei-Variable noch nicht (durch einen Aufruf von *Assign*) einer Datei zugeordnet wurde.

103 File not open
(Datei ist nicht offen)
Ausgelöst von *Close*, *Read*, *Write*, *Seek*, *Eof*, *FilePos*, *FileSize*, *Flush*, *BlockRead* und *BlockWrite* bei der Anwendung auf eine Datei, die noch nicht eröffnet wurde.

104 File not open for input
(Datei wurde nicht für Leseoperationen geöffnet)
Ausgelöst von *Read*, *Readln*, *Eof*, *Eoln*, *SeekEof* und *SeekEoln* bei der Anwendung auf eine Textdatei, die nur für Schreiboperationen geöffnet wurde.

105 File not open for output
(Datei wurde nicht für Schreiboperationen geöffnet)
Ausgelöst von *Write*, *Writeln* und *Page* bei der Anwendung auf eine Textdatei, die nur für Leseoperationen geöffnet wurde.

106 Invalid numeric format
(ungültiges numerisches Format)
Ausgelöst von *Read* oder *Readln*, wenn von einer Textdatei gelesene Werte nicht dem numerischen Format der angegebenen Variablen entsprechen.

Fehler mit sofortigem Abbruch (»Fatal Errors«)

200 Division by zero
(Division durch Null)

201 Range check error
(Bereichsüberprüfung: Fehler entdeckt)
Diesen Fehler erhalten Sie, wenn das Programm im Modus {$R+} compiliert wurde und eine der folgenden Bedingungen zutrifft:

- der Index-Ausdruck bei einem Zugriff auf ein Array gibt ein nicht existierendes Element an (d.h. sein Wert überschreitet die Array-Grenzen);
- das Programm hat versucht, einer Variablen einen Wert außerhalb ihres zulässigen Bereichs zuzuweisen;
- das Programm hat versucht, einer Prozedur oder Funktion einen Wert zu übergeben, der außerhalb des zulässigen Wertebereichs des entsprechenden formalen Parameters dieser Routine liegt.

202 Stack overflow error
(Stacküberprüfung: Überlauf entdeckt)
Diesen Fehler erhalten Sie, wenn das Programm im Modus {$S+} compiliert wurde und es sich nun beim Aufruf einer Prozedur oder Funktion herausstellt, daß nicht mehr genügend Platz auf dem Stack vorhanden ist. (Solange es sich nicht um einen Programmierfehler handelt: verwenden Sie den Compiler-Befehl {$M}, um die Größe des Stacks entsprechend zu erhöhen.)

203 Heap overflow error
(kein Platz mehr im Heap-Bereich)
Dieser Fehler wird von *New* und *GetMem* ausgelöst, wenn nicht mehr genug Platz auf dem Heap ist, um einen Speicherbereich der gewünschten Größe zu

belegen. (Detaillierte Informationen, auch über das Abfangen solcher Fehler, finden Sie in Kapitel 16 »Speicherbelegung und Datenformate«).

204 Invalid pointer operation
(ungültige Zeiger-Operation)
Dieser Fehler wird von *Dispose* oder *FreeMem* erzeugt, wenn die übergebene Zeigervariable den Wert **nil** hat oder auf eine Adresse außerhalb des Heap zeigt.

205 Floating point overflow
(Gleitkomma-Überlauf)

206 Floating point underflow
(Gleitkomma-Unterlauf)
Unterläufe liefern normalerweise den Wert 0 diesen Fehler erhalten Sie nur bei Verwendung eines numerischen Coprozessors und der Freigabe des Bits für Unterläufe im Statusregister des 8087.

207 Invalid floating point operation
(Gleitkomma-Fehler)

- Ausgelöst von *Trunc* und *Round*, wenn der übergebene Real-Wert nicht innerhalb der Bereichsgrenzen von *LongInt* (-2147483648.. 2147483647) in einen ganzzahligen Wert konvertiert werden konnte.
- Ausgelöst von *Sqrt* bei Aufrufen mit negativem Argument.
- Ausgelöst von *Ln* bei Aufrufen mit negativem Argument oder dem Wert 0.
- Ausgelöst durch einen Stack-Überlauf des 8087 (mehr als acht Zwischenergebnisse bzw. Überlauf durch Rekursion siehe Kapitel 15 »Der Coprozessor 8087«).

210 Object not initialized.
[Objekt nicht initialisiert.]
Sie haben versucht, die virtuelle Methode eines Objektes aufzurufen, obwohl dieses Objekt noch nicht (mittels eines Konstruktor-Aufrufs) initialisiert ist.

211 Call to abstract method
(Aufruf einer abstrakten Methode)
Dieser Fehler wird von der Prozedur *Abstract* in der Unit *WObjects* ausgelöst und zeigt an, daß Ihr Programm versucht hat, eine abstrakte virtuelle Methode aufzurufen. Enthält ein Objekt eine oder mehrere abstrakte Methoden, dann handelt es sich bei diesem Objekt um einen *abstrakten Objekttyp*, der nicht instantiiert werden darf. Sie müssen ein neues Objekt von diesem abstrakten Objekttyp ableiten und dessen abstrakte Methoden überschreiben. Zum Beispiel ist die Methode *Compare* des Objekts *TSortedCollection* in der Unit *WObjects* abstrakt. Sie müssen also von *TSortedCollection* ein neues Objekt ableiten und die Methode *Compare* überschreiben.

212 Stream registration error
(Fehler bei Registrierung eines Objekts)
Dieser Fehler wird von der Prozedur *RegisterType* in der Unit *Objects* ausgelöst und zeigt an, daß einer der folgenden Fehler vorliegt:

- Der Stream-Registrationsrecord befindet sich nicht im Datensegment.
- Das Feld *ObjType* des Stream-Registrationsrecords enthält den Wert Null.
- Der Typ wurde schon registriert.
- Es existiert schon ein anderer Typ mit dem gleichen Wert in *ObjType*.

213 Collection index out of range
(Index außerhalb des zulässigen Bereichs)
Der Index, der beim Aufruf einer Methode von *TCollection* übergeben wurde, liegt außerhalb des zulässigen Bereichs.

214 Collection overflow error
(Überlauf einer Kollektion)
Dieser Fehler wird von *TCollection* ausgelöst, wenn Sie versuchen, einer Kollektion, die nicht mehr erweitert werden kann, ein Element hinzuzufügen.

B. Zeichensatz und Tastaturcodes

ASCII-Codes

Intern stellt der Computer Zeichen (wie allen anderen Daten auch) durch Zahlenwerte dar. Der Zusammenhang zwischen Zeichen und dazugehörigen Nummern wird durch den *American Standard Code for Information Interchange* (= amerikanischer Standard-Code zum Informationsaustausch, ASCII) festgelegt. Ursprünglich war dieser Code nur mit 7 Bit (= 128 Zeichen) definiert – die Firma IBM hat sich die Freiheit genommen, ihn um ein zusätzliches Bit zu erweitern (d.h. insgesamt auf 256 Zeichen). Dieser auf dem PC übliche Zeichensatz ist allerdings unter Windows nicht verfügbar; hier gilt der ANSI-Zeichensatz.

Die Tabelle B.1 auf den nächsten beiden Seiten gibt die Zeichen des ASCII-Codes sowie seiner Erweiterung durch IBM wieder.

Tabelle B.2 listet den unter Windows verfügbaren Zeichensatz auf. Die nicht mit Zeichen belegten Stellen sind in diesem Zeichensatz nicht definiert.

Tabelle B.1: Der ASCII-Zeichensatz und Erweiterungen

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
0	0		43	2B	+	86	56	V
1	1	☺	44	2C	,	87	57	W
2	2	☻	45	2D	-	88	58	X
3	3	♥	46	2E	.	89	59	Y
4	4	♦	47	2F	/	90	5A	Z
5	5	♣	48	30	0	91	5B	[
6	6	♠	49	31	1	92	5C	\
7	7	•	50	32	2	93	5D	]
8	8	◘	51	33	3	94	5E	^
9	9	○	52	34	4	95	5F	_
10	A	◙	53	35	5	96	60	‘
11	B	♂	54	36	6	97	61	a
12	C	♀	55	37	7	98	62	b
13	D	♪	56	38	8	99	63	c
14	E	♫	57	39	9	100	64	d
15	F	¤	58	3A	:	101	65	e
16	10	▶	59	3B	;	102	66	f
17	11	◀	60	3C	<	103	67	g
18	12	↕	61	3D	=	104	68	h
19	13	‼	62	3E	>	105	69	i
20	14	¶	63	3F	?	106	6A	j
21	15	§	64	40	@	107	6B	k
22	16	■	65	41	A	108	6C	l
23	17	↨	66	42	B	109	6D	m
24	18	↑	67	43	C	110	6E	n
25	19	↓	68	44	D	111	6F	o
26	1A	→	69	45	E	112	70	p
27	1B	←	70	46	F	113	71	q
28	1C	∟	71	47	G	114	72	r
29	1D	↔	72	48	H	115	73	s
30	1E	▲	73	49	I	116	74	t
31	1F	▼	74	4A	J	117	75	u
32	20		75	4B	K	118	76	v
33	21	!	76	4C	L	119	77	w
34	22	"	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	$	79	4F	O	122	7A	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	|
39	27	’	82	52	R	125	7D	}
40	28	(	83	53	S	126	7E	~
41	29	)	84	54	T	127	7F	Δ
42	2A	*	85	55	U	128	80	Ç

Tabelle B.1: Der ASCII-Zeichensatz und Erweiterungen (Fortsetzung)

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
129	81	ü	172	AC	¼	214	D6	╓
130	82	é	173	AD	¡	215	D7	╫
131	83	â	174	AE	«	216	D8	╪
132	84	ä	175	AF	»	217	D9	┘
133	85	à	176	B0	▒	218	DA	┌
134	86	å	177	B1	▒	219	DB	▉
135	87	ç	178	B2	▓	220	DC	▄
136	88	ê	179	B3	│	221	DD	▌
137	89	ë	180	B4	┤	222	DE	▐
138	8A	è	181	B5	╡	223	DF	▀
139	8B	ï	182	B6	╢	224	E0	α
140	8C	î	183	B7	╖	225	E1	β
141	8D	ì	184	B8	╕	226	E2	Γ
142	8E	Ä	185	B9	╣	227	E3	π
143	8F	Å	186	BA	║	228	E4	Σ
144	90	É	187	BB	╗	229	E5	σ
145	91	æ	188	BC	╝	230	E6	µ
146	92	Æ	189	BD	╜	231	E7	τ
147	93	ô	190	BE	╛	232	E8	ϕ
148	94	ö	191	BF	┐	233	E9	θ
149	95	ò	192	C0	└	234	EA	Ω
150	96	û	193	C1	┴	235	EB	δ
151	97	ù	194	C2	┬	236	EC	∞
152	98	ÿ	195	C3	├	237	ED	∅
153	99	Ö	196	C4	─	238	EE	ε
154	9A	Ü	197	C5	┼	239	EF	∩
155	9B	¢	198	C6	╞	240	F0	≡
156	9C	£	199	C7	╟	241	F1	±
157	9D	¥	200	C8	╚	242	F2	≥
158	9E	₧	201	C9	╔	243	F3	≤
159	9F	*f*	202	CA	╩	244	F4	⌠
160	A0	á	203	CB	╦	245	F5	⌡
161	A1	í	204	CC	╠	246	F6	÷
162	A2	ó	205	CD	═	247	F7	≈
163	A3	ú	206	CE	╬	248	F8	°
164	A4	ñ	207	CF	╧	249	F9	•
165	A5	Ñ	208	D0	╨	250	FA	·
166	A6	ª	209	D1	╤	251	FB	√
167	A7	º	210	D2	╥	252	FC	^{n}
168	A8	¿	211	D3	╙	253	FD	^{2}
169	A9	⌐	212	D4	╘	254	FE	■
170	AA	¬	213	D5	╒	255	FF	
171	AB	½	214	D6	╓			

Tabelle B.2: Der ANSI-Windows-Zeichensatz

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
0	0		43	2B	+	86	56	V
1	1		44	2C	,	87	57	W
2	2		45	2D	-	88	58	X
3	3		46	2E	.	89	59	Y
4	4		47	2F	/	90	5A	Z
5	5		48	30	0	91	5B	[
6	6		49	31	1	92	5C	\
7	7		50	32	2	93	5D	]
8	8		51	33	3	94	5E	^
9	9		52	34	4	95	5F	_
10	A		53	35	5	96	60	‘
11	B		54	36	6	97	61	a
12	C		55	37	7	98	62	b
13	D		56	38	8	99	63	c
14	E		57	39	9	100	64	d
15	F		58	3A	:	101	65	e
16	10		59	3B	;	102	66	f
17	11		60	3C	<	103	67	g
18	12		61	3D	=	104	68	h
19	13		62	3E	>	105	69	i
20	14		63	3F	?	106	6A	j
21	15		64	40	@	107	6B	k
22	16		65	41	A	108	6C	l
23	17		66	42	B	109	6D	m
24	18		67	43	C	110	6E	n
25	19		68	44	D	111	6F	o
26	1A		69	45	E	112	70	p
27	1B		70	46	F	113	71	q
28	1C		71	47	G	114	72	r
29	1D		72	48	H	115	73	s
30	1E		73	49	I	116	74	t
31	1F		74	4A	J	117	75	u
32	20	space	75	4B	K	118	76	v
33	21	!	76	4C	L	119	77	w
34	22	"	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	$	79	4F	O	122	7A	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	|
39	27	’	82	52	R	125	7D	}
40	28	(	83	53	S	126	7E	~
41	29	)	84	53	T	127	7F	
42	2A	*	85	54	U	128	80	

Tabelle B.2: Der ANSI-Zeichensatz (Fortsetzung)

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
129	81		172	AC		215	D7	
130	82		173	AD		216	D8	
131	83		174	AE		217	D9	
132	84		175	AF		218	DA	
133	85		176	B0		219	DB	
134	86		177	B1		220	DC	
135	87		178	B2		221	DD	
136	88		179	B3		222	DE	
137	89		180	B4		223	DF	
138	8A		181	B5		224	E0	
139	8B		182	B6		225	E1	
140	8C		183	B7		226	E2	
141	8D		184	B8		227	E3	
142	8E		185	B9		228	E4	
143	8F		186	BA		229	E5	
144	90		187	BB		230	E6	
145	91		188	BC		231	E7	
146	92		189	BD		232	E8	
147	93		190	BE		233	E9	
148	94		191	BF		234	EA	
149	95		192	C0		235	EB	
150	96		193	C1		236	EC	
151	97		194	C2		237	ED	
152	98		195	C3		238	EE	
153	99		196	C4		239	EF	
154	9A		197	C5		240	F0	
155	9B		198	C6		241	F1	
156	9C		199	C7		242	F2	
157	9D		200	C8		243	F3	
158	9E		201	C9		244	F4	
159	9F		202	CA		245	F5	
160	A0		203	CB		246	F6	
161	A1		204	CC		247	F7	•
162	A2		205	CD		248	F8	
163	A3		206	CE		249	F9	
164	A4		207	CF		250	FA	
165	A5		208	D0		251	FB	
166	A6		209	D1		252	FC	
167	A7		210	D2		253	FD	
168	A8		211	D3		254	FE	
169	A9		212	D4		255	FF	
170	AA		213	D5				
171	AB		214	D6				

Scancodes der Tastatur

Die folgende Tabelle listet (in hexadezimaler Notation) die Scancodes auf, die von der Tastatur zurückgeliefert werden.

Tabelle B.3 : Scancodes der Tastatur

Taste	Scan Code in Hex	Taste	Scan Code in Hex
Esc	01	Tabulator	0F
!1	02	Q	10
"2	03	W	11
§3	04	E	12
$4	05	R	13
%5	06	T	14
&6	07	Z	15
/7	08	U	16
(8	09	I	17
)9	0A	O	18
=0	0B	P	19
?ß	0C	Ü	1A
‘’	0D	*+	1B
Backspace	0E	Return	1C
Ctrl	1D	^#	2B
A	1E	Y	2C
S	1F	X	2D
D	20	C	2E
F	21	V	2F
G	22	B	30
H	23	N	31
J	24	M	32
K	25	;,	33
L	26	:.	34
Ö	27	_-	35
Ä	28	Shift-rechts	36
><	29	Prtsr*	37
Shift-links	2A	Alt	38
Leertaste	39	7 Home	47
Caps Lock	3A	8 Aufwärtspfeil	48
F1	3B	9 PgUp	49
F2	3C	Minuszeichen	4A
F3	3D	4 Linkspfeil	4B
F4	3E	5	4C

Taste	Scan Code in Hex	Taste	Scan Code in Hex
F5	3F	6 Rechtspfeil	4D
F6	40	+	4E
F7	41	1 End	4F
F8	42	2 Abwärtspfeil	50
F9	43	3 PgDn	51
F10	44	0 Ins	52
F11	D9	Del	53
F12	DA	NumLock	45
ScrollLock	46		

Index

A
Abs (Funktion) 295
Abschneiden
  von Dateien 367
  von Nachkommastellen 367
absolute Variablen 51
absolute Werte 295
Addr (Funktion) 296
Adreß-Ermittlung CSeg-Funktion 305
  DSeg-Funktion 309
  Ofs-Funktion 334
  Seg-Funktion 349
  SSeg 355
Adreß-Operator 77
Adreßübergabe 223
Aktivierung
  mit vollständgem Bezeichner 87
Align data (Compiler-Schalter) 244
Anhängen von Daten 296
Anweisung 85
  Prozedur 87
  with 55, 96
  bedingte 90
  case 91
  einfache 85
  goto 88
  if 90
  Prozeduraufruf 87
  strukturierte 89
  Verbund- 89
  Wiederholung 92
  Zuweisungen 86
Append (Prozedur) 296
ArcTan (Funktion) 297
Arrays
  als Parameter 225
  Speicherung 201
  Indizierung
  Compiler-Schalter 248
  Typen 31
  typisierte Konstanten 61
  und Prozedur-Variablen 115
ASCII-Codes 401
ASCIIZ-Format 202
Assembler-Code
  mit dem integrierten Assembler 259
  und Turbo Pascal 283
  Deklaration 104
Assign (Prozedur) 298
AssignCrt (Prozedur) 299
Attribute von Dateien 154
Aufruf
  dynamischer Methoden 212
  virtueller Methoden 211
  Konventionen 87
  mit vollständigem Bezeichner 87
  near und far 225
  von Funktionen und Prozeduren 223
  von Funktionen 80
  von Prozeduren 87
  von Unterprogrammen 110f.
  Methode 213
Aufzählungstypen 27
  als Parameter 224
  Speicherung 197
Ausdrücke 67
  absolute 275
  Konstanten-Vorausberechnung 237
  Mengen-Konstruktoren 81
  Operatoren 67, 71ff., 74f.
  Reihenfolge der Berechnung 239
  relozierbare 275
  Syntax 67
  Umwandlung von Wert-Typen 81
  und Prozedur-Variablen 82
Ausgabe
  Spezialzeichen 172
  Compiler-Schalter I/O-checking 247

Laufzeitfehler 397
Write-Prozedur 370
Ausschluß
unbenutzter Daten 241
unbenutzter Programmteile 241
Austritts-Code 280
Auswertung von Ausdrücken
Compiler-Schalter 74, 238
automatische Konvertierung
von -Formaten 26

B
bedingte
Anweisungen 90
Compilierung 253f.
Berechnung
Reihenfolge 239
Bereichs-Überprüfung
(Compiler-Schalter) 248
Bezeichner 10
Gültigkeitsbereich 19f.
Methode 40
qualifizierte 10
Standard- 10
vollständige mit Self 88
Bildschirm
Prozeduren und Funktionen 176
AssignCrt-Prozedur 299
Löschen bis Zeilenende 303
Bindung
dynamische 38
Blöcke 17
Gültigkeitsbereich (Scope) 19
Syntax 17
BlockRead (Prozedur) 299
BlockWrite (Prozedur) 300
Boolean (Datentyp) 26
als Parameter 224
Speicherung 197
Ausgabe 371
Boolean evaluation (Compiler-Schalter)
245
boolesche Operatoren 73
Byte (Datentyp)
als Parameter 224
Speicherung 196

C
case-Anweisung 91
Char (Datentyp) 26
als Parameter 224
Speicherung 196
Ausgabe 371
Lesen aus Dateien 339
ChDir (Prozedur) 301
Chr (Funktion) 302
Close (Prozedur) 302
Close-Funktion bei Gerätetreibern 235
ClrEol (Prozedur) 303
ClrScr (Prozedur) 303
CODE (Segmentname) 283
Comp (Datenformat) 183
Comp (Datentyp)
als Parameter 224
Speicherung 200
Compiler
automatische Optimierung 237
Parameter 251
Schalter 238
Compiler-Anweisung
$B 74
$A 240
$L 216
$N 29
$R 207
$S 50
bedingte Compilierung 253
Concat (Funktion) 303
CONST (Segmentname) 283
constructor (Schlüsselwort) 107
Coprozessor
8087 (Coprozessor) 181
Ausgabe von Realzahlen 186
Datentypen 182
Rechenstack 185
und Units 186
Compiler-Schalter 252
Copy (Funktion) 304
Cos (Funktion) 304
CSeg (Funktion) 305
CSEG (Segmentname) 283
Ctrl-Break
DOS-Flag 349
Cursor
im Textmodus 370

D
DATA (Segmentname) 283
Datei-Typen 41
Speicherung 201
Dateien
Attribute 154, 322
blockweise Lesen 299
blockweise Schreiben 300

Compiler-Schalter I/O checking 247
Close-Funktion 145, 235
Dateiende prüfen 309, 347
Datum und Uhrzeit 323, 351
Eröffnen 145
Flush-Funktion 235
InOut-Funktion 234
Modus-Konstanten 154
Öffnungsmodus 149
Open-Funktion 234
Position 313, 347
Schließen 145, 302
Schreiben 370
Schreiben erzwingen 318
Suche nach 317
Textpuffer setzen 352
Typen und Records 155
Umbenennen 342
Zeilenende prüfen 310, 348
Datensegment 50
Datentypen des Coprozessors 182
Datum und Uhrzeit
Prozeduren 157
GetDate-Prozedur 321
GetFTime-Prozedur 323
GetTime-Prozedur 324
PackTime-Prozedur 335
SetDate-Prozedur 350
SetFTime-Prozedur 351
SetTime-Prozedur 353
UnpackTime-Prozedur 368
Debug information (Compiler-Schalter) 245
Dec (Prozedur) 305
DEFINE (bedingter Compiler-Befehl) 255
Deklaration
Methode 37, 107
Objekttypen 35
Funktionen 105
Prozedurtypen 43
Variablen 49
Prozeduren 99
Delete (Prozedur) 306
destructor (Schlüsselwort) 107
Destruktor 107
Aufrufkonventionen 213
Definition 108
Dezimalpunkt, impliziter 197
Disketten- und Festplattenlaufwerke
Statusfunktionen 158
DiskFree (Funktion) 306
DiskSize (Funktion) 307
Dispose 142
Destruktor als Parameter 213
erweiterte Syntax 39, 108, 214
Prozedur 307
VMT 207
DMT 208
DOS
Funktionsaufrufe 332
Geräteeinheiten 149
Parameter ermitteln/setzen 159
DosError (Variable) 157
DosVersion (Funktion) 308
Double (Datentyp) 182
als Parameter 224
Speicherung 199
DSeg (Funktion) 309
DSEG (Segmentname) 283
dynamische Bindung 38
dynamische Methode
Aufruf 212
dynamische Objektinstanzen 213
dynamische Speicherverwaltung
Dispose-Prozedur 307
Fehlerbehandlung 195
FreeMem-Prozedur 319
GetMem-Prozedur 324
MaxAvail-Funktion 330
MemAvail-Funktion 331
New-Prozedur 333

E
Einbinden von Object-Dateien (Compiler-Parameter) 251
Eingabe
Compiler-Schalter I/O checking 247
direktes Lesen der Tastatur 341
KeyPressed-Funktion 328
Laufzeitfehler 397
Read-Prozedur 339
Readln-Prozedur 342
Steuerzeichen 173
zeilenweise 173
Eintritts-Code 280
ELSE (bedingter Compiler-Befehl) 256
Emulation des Coprozessors
in Maschinenprogrammen 187
ENDIF (bedingter Compiler-Befehl) 256
Eof(Funktion) 309
Eoln(Funktion) 310
Erase (Prozedur) 311
ErrorAddr (Variable) 230

Ersatzmethode 38
Erweiterte Syntax
  Compiler-Schalter 249
  Funktionsanweisungen 249
Exit (Prozedur) 311
Exit-Prozeduren 229
ExitCode (Variable) 230
ExitProc (Variable) 229
Exp (Funktion) 312
Extended (Datenformat) 183
  als Parameter 224
  Ausgaben 186
  Operationen mit 183
  Speicherung 199
external (Schlüsselwort) 216
external-Deklaration 103

F
Fail (Standardprozedur) 220
far-Aufrufe 225
  Compiler-Schalter 246
  Deklaration 100
  und Prozedur-Variablen 114
Fehler
  bei numerischen Vergleichen 184
  Laufzeit- 395
  mit sofortigem Abbruch 398
  Syntax- 377
Fehlerbehandlung
  Aufruf virtueller Methoden 207
  Compiler-Schalter I/O-checking 247
  für den Heap 195
  Fail 220
  HeapError 219
  in Konstruktoren 219
  IOResult-Funktion 328
  Variablen ErrorAddr und ExitCode 230
Fehlermeldungen und -codes 377
Fehlersuche
  Compiler-Schalter Debug information 245
  lokale Symbole 247
FExpand (Funktion) 312
FileMode (Variable) 149
FilePos (Funktion) 313
FileRec (Datentyp) 155, 202
FileSize (Funktion) 314
FillChar(Prozedur) 316
FindFirst (Prozedur) 317
FindNext (Prozedur) 318
Flush (Prozedur) 318
Flush-Funktion bei Gerätetreibern 235
for-Anweisung 94
Force far calls (Compiler-Schalter) 246
forward-Deklaration 102
Frac (Funktion) 319
FreeMem (Prozedur) 319
Füllen von Speicherbereichen 316
Funktion
  SizeOf 278
Funktionen
  Aufruf und Parameterübergabe 223
  Deklaration als Inline 291
  eingeschachtelte 226
  Ergebnisse 225
  Maschinensprache 283
  SeekEoln 348
  Start- und Endcode 227
  Abbruch der Ausführung 311
Funktionsergebnis
  Rückgabe 280
Funktionsergebnis verwerfen 249

G
Generate 80286 Code (Compiler-Schalter) 246
Geräte 149
Gerätetreiber
  Close-Funktion 235
  Flush-Funktion 235
  für Textdateien 233
  InOut-Funktion 234
  Open-Funktion 234
GetCBreak (Prozedur) 320
GetDate (Prozedur) 321
GetDir (Prozedur) 321
GetEnv (Funktion) 321
GetFAttr (Prozedur) 322
GetFTime (Prozedur) 323
GetIntVec (Prozedur) 323
GetMem (Prozedur) 324
  Fehlerbehandlung 196
GetTime (Prozedur) 324
GetVerify (Prozedur) 324
Gleitkomma
  Compiler-Schalter 252
  Datentypen 182
  Dezimalpunkt 197
  Konstanten
  Operationen 29
  Speicherung 183
  werte 181
goto-Anweisung 88
Gültigkeitsbereich 107

H
Halt (Prozedur) 325
Heap-Bereich
  Compiler-Parameter 252
HeapError (Variable) 195, 219
Hi (Funktion) 325

I
I/O checking (Compiler-Schalter) 247
I/O-Adressen
  direkter Zugriff 236
I/O-Fehler siehe Laufzeitfehler: 397
identische Typen 44
if-Anweisung 90
IFDEF (bedingter Compiler-Befehl) 255
IFNDEF (bedingter Compiler-Befehl) 256
IFOPT (bedingter Compiler-Befehl) 256
Implementierung, Methode 37
Inc (Prozedur) 325
Include-Datei (Compiler-Parameter) 251
Indizierungen 53
  Index-Prüfung 239
  Optimierung 240
inline 289
  Anweisungen 289
  Deklarationen 104, 291
  und Prozedur-Variablen 114
InOut-Funktion bei Gerätetreibern 234
Insert (Prozedur) 326
Instanz
  Objekt 38
Int (Funktion) 327
Integer (Datentyp) 25
  als Parameter 224
  Ausgabe 371
  Lesen aus Dateien 339
  Speicherung 196
interrupt
  Deklaration 101
  und Prozedur-Variablen 114
Interrupts
  Ausführen 327
  Bearbeitung 231
  DOS 332
  Hardware 232
  Prozeduren 158, 231
  Vektoren ermitteln 323
  Vektoren setzen 351
Intr (Prozedur) 327
IOResult (Funktion) 328
  Ergebnisse 397

K
KeyPressed (Funktion) 328
Kommandozeilen-Parameter
  Anzahl 335
  Lesen 335
Kommentare 16, 260
kompatible Typen 45
Konstanten 14, 59
  berechnete 14
  einfach typisiert 60
  Fließkomma 183
  gemeinsame Speicherung 237
  in der Unit WinDos 153
  Optimierung bei Multiplikationen 240
  von Objekten 63
  Vorausberechnung 237
Konstruktor 107
  Aufrufkonventionen 213
  Definition 108
  Fail (Fehlerbehandlung) 220
  Fehlerbehandlung 219
  und VMT 205, 208
  vererbter 38
Konvertierung
  in Großbuchstaben 368
  numerische Werte in Strings 355
  von Formaten 26
  von Strings in numerische Werte 368
Kopieren von Speicherbereichen 332
Kurzschlußverfahren 73, 238
Compiler-Schalter 245

L
Labels 12, 261
  Bezeichner lokaler 261
  lokale 261
  normale 262
Laufzeitfehler 395
  direkt erzeugen 346
Length (Funktion) 329
Linker
  unbenutzte Programmteile 241
Ln (Funktion) 329
Lo (Funktion) 329
Local symbols (Compiler-Schalter) 247
logische Operatoren 72
lokale Compiler-Schalter 244
lokale Variablen 50
  Speicherung 291
LongBool (Datentyp) 26
  Speicherung 197

LongInt (Datentyp)
  als Parameter 224
  Speicherung 196

M
Maschinencode
  mit Inline 289
Maschinenprogramme
  mit dem integrierten Assembler 259
  und der Emulator 187
  und Turbo Assembler 284
mathematischer Coprozessor 181
MaxAvail (Funktion) 330
Mem, MemW, MemL (Pseudo-Arrays) 203
MemAvail (Funktion) 331
Memory sizes (Compiler-Parameter) 252
Mengen
  als Parameter 225
  Konstruktoren 81
  Operatoren 75
  Speicherung 201
  typisierte Konstanten 64
  Vergleiche 77
Methode
  Aufruf dynamischer 212
  Aufruf statischer 40
  Aufruf virtueller 40, 211
  Aufrufkonventionen 213
  Bezeichner 40, 87
  Definition 34
  Deklaration 37, 107
  Ersatzmethode 38
  externe 216
  Gültigkeitsbereich 107
  Implementierung 37, 107
  in Assembler 216
  Kompatibilität von Parametern 111
  Parameter 87
  statisch 38
  virtuell 38, 87
  vollständiger Bezeichner 37, 87
  Zugriff über vollständige Bezeichner 55
MkDir (Prozedur) 331
Move (Prozedur) 332
MSB 197
MsDos (Prozedur) 332
Multiplikations-Optimierung 240

N
Nachkomme 35
natürlicher Logarithmus 312, 329
near-Aufrufe 225
  Deklaration 100
New 42, 142
  erweiterte Syntax 38, 108, 214
  Fehlerbehandlung 196
  Konstruktor als Parameter 213
  Prozedur 333
  VMT 207
nil (Zeiger-Konstante)
  Wert 200
Nullstring 30
Numeric processing (Compiler-Schalter) 252
numerische Werte 13
  und der 8087 186
numerischer Coprozessor 181

O
Objekt
  Bereich 35
  Definition eines Destruktors 108
  Definition eines Konstruktors 108
  DMT 208
  dynamische Freigabe 108
  dynamische Instanz 213
  dynamische Reservierung 108
  Feld 34
  Format 205
  Gültigkeitsbereich 37
  Gültigkeitsbereich von Feldern 37
  Instanz 38
  internes Datenformat 205
  Konstruktor 38
  Konstruktoren und VMT 108, 205, 208
  Nachkomme 35
  Polymorphie 40, 86, 111
  Typ 34
  Typdeklaration 35
  typisierte Konstanten 63
  Vererbung 35
  VMT 206
  VMT und Zeigerinitialisierung 208
  Zeiger auf 40
Odd (Funktion) 334
Ofs (Funktion) 334
Open-Funktion bei Gerätetreibern 234
Operatoren 71
  boolesche 73
  logische 72
  Rangfolge 277
  Rangfolge 67
  relationale 76

Syntax 67
unäre 67
Optimierung
automatische 237
Optimierungen 279
Ord (Funktion) 334
ordinale Typen 24
Pred-Funktion 337
Succ-Funktion 366

P
Pack 142
PackTime (Prozedur) 335
ParamCount (Funktion) 335
Parameter 109, 111
Adressübergabe 223
Compiler- 251
prozedurale 115
Self 87f., 107, 213
Typkompatibilität 111
untypisierte 111
Variablen- 110
VMT 213
Wert- 110
Wertübergabe 224
ParamStr (Funktion) 335
PChar (Datentyp) 42
Peripheriegeräte 149
Pi (Funktion) 336
Pointer (Datentyp) 42
Polymorphie
Typkompatibilität von Parametern 111
Zeigerzuweisung 40
Zuweisung einer Objektinstanz 86
Port, PortW (Pseudo-Arrays) 236
Pos (Funktion) 336
Präfix-Opcodes 262
Pred (Funktion) 337
Programm
Ausschluß unbenutzter Teile 241
Programmcode
automatische Optimierung 237
toter 240
Programme
Ausführung abbrechen 325
Prozedur-Aufruf
effizienter 277
Prozedur-Variablen 112
als Parameter 115
in Ausdrücken 82
Speicherplatzbedarf 113
Typ-Umwandlung 57
Prozedur
Abbruch der Ausführung 311
Adreermittlung 296
Anweisungen 87
Aufruf 87
Deklaration als Inline 291
eingeschachtelte 226
Interrupts 231
Maschinensprache 283
Parameterübergabe 223
Start- und Endcode 227
zum Beenden eines Programms 229
prozedurale Typen 43, 112
Deklaration 43
Speicherung 203
typisierte Konstanten 65
Prozessor-Register
für Funktionsergebnisse 225
verfügbare 229
Prüfung
von Wertebereichen 239
des Stacks 50
Compiler-Schalter 247f.
Ptr (Funktion) 337
PUBLIC-Bezeichner 283

Q
Quadratwurzeln 355
Quadrierung 355
Qualifizierer 52
qualifizierte Bezeichner 10

R
Rad (Winkelmaß) 304
Random (Funktion) 338
Randomize (Prozedur) 338
RandSeed (Variable) 151, 338
Range checking (Compiler-Schalter) 248
Read (Prozedur) 339
ReadKey (Funktion) 341
Readln (Prozedur) 342
Real (Datentyp) 181
als Parameter 224
Ausgabe 371
Lesen aus Dateien 339
Speicherung 197f.
und Unendlichkeiten 198
Realzahlen 181
8087-Berechnung 29
Ausgabe mit dem 8087 186
Compiler-Schalter 252
Extended 183

Software-Berechnung 29
Vergleiche 184
Rechenstack des Coprozessors 185
Records
als Parameter 225
für Dateien 155
Speicherung 201
typisierte Konstanten 62
und Feldbezeichner 54
und Prozedur-Variablen 115
Register
Einsatz 260
Flags 153
Reihenfolge der Berechnung 239
relationale Operatoren 76
Relokation 275
Rename (Prozedur) 342
repeat-Anweisung 93
reservierte Wörter 8
Reset (Prozedur) 343
Öffnungsmodi für Dateien 149
RET-Befehl 263
Rewrite (Prozedur) 344
Öffnungsmodus für Dateien 149
RmDir (Prozedur) 345
Round (Funktion) 346
RunError (Prozedur) 346

S
Schalter, für den Compiler 244
Schiebebefehle anstelle von Multiplikationen 240
Schleifen
for 94
repeat..until 93
while 93
Seek (Prozedur) 347
SeekEof (Funktion) 347
SeekEoln (Funktion) 348
Seg (Funktion) 349
SEG-Operator 271
Segmentnamen
Code-Segment 283
Daten-Segmente 283
Self 88, 107
in Methoden 87
Parameter 213
Parameter in Konstruktoren und Destruktoren 213
SetCBreak (Prozedur) 349
SetDate (Prozedur) 350
SetFAttr (Prozedur) 350
SetFTime (Prozedur) 351
SetIntVec (Prozedur) 351
SetTextBuf (Prozedur) 352
SetTime (Prozedur) 353
SetVerify (Prozedur) 353
ShortInt (Datentyp)
als Parameter 224
Speicherung 196
Sin (Funktion) 353
Single (Datentyp)
als Parameter 224
Datenformat 182
Speicherung 198
SizeOf (Funktion) 211, 354
Speicher
direkter Zugriff 203
Speicher-Ausdruck
untypisiert 276
Speicherbereiche
füllen 316
Größe 354
Spezialzeichen
Ausgabe über WinCrt 172
Erzeugen mit chr-Funktion 302
Sprachelemente 7
Sprunganpassung
automatische 264
SPtr (Funktion) 354
Sqr (Funktion) 355
Sqrt (Funktion) 355
SSeg (Funktion) 355
Stack checking (Compiler-Schalter) 248
Stack-Rahmen 280
Stack-Segment
ermitteln 355
Stack-Überlauf
automatische Prüfung 50
Stack
Größe, Compiler-Parameter 252
Prüfung 248
Stacksegment 50
Start- und Endcode 227
Statement 85
statische Methode 38
Aufruf 40
Str (Prozedur) 355
für das Format Extended 186
String (Datentyp)
Absuchen 336
als Parameter 225
als var-Parameter, Compiler-Schalter 248

Ausgabe 371
Einfügen 326
Indizierungen, Compiler-Schalter 248
Konstanten 269
Konvertierung 355, 368
Länge bestimmen 329
Löschen 306
Lesen aus Dateien 339
nullterminierter 42
Operatoren 74
Speicherung 200
Typen 13, 30
typisierte Konstanten 60
Verbinden 303
Vergleiche 76
strukturierte Typen 30
typisierte Konstanten 61
Succ (Funktion) 366
Swap (Funktion) 366
Symbole für bedingte Compilierung 254
Syntax, erweiterte 42, 45f., 62, 65
Syntaxdiagramm, Begriffsdefinition 10
System (Unit)
und Prozedur-Variablen 114

T
TASM 284
Tastatur
direkte Prüfung 328
direktes Lesen 341
TDateTime (Record-Typ) 156
Teilbereichstyp 27
Textdatei-Gerätetreiber 150, 233
Textpuffergröße setzen 352
TextRec, Datentyp 155, 202
Tokens 7
toter Code, automatische Entfernung 240
TRegisters, Datentyp 156, 327, 332
Trigonometrie
ArcTan-Funktion 297
Cos-Funktion 304
Pi-Funktion 336
Sin-Funktion 353
Trunc (Funktion) 367
Truncate (Prozedur) 367
TSearchRec (Record-Typ) 156
Turbo Assembler 284
Typ-Bezeichner 273
Typ-Prüfung 275
Typ-Umwandlung
von Prozedur-Variablen 57
von Werten 81
Typangabe 276
Typdeklaration
Objekt 35
Typen 23
Definition 46
einfache 24
identische 44
in der Unit WinDos 153
kompatible 45
ordinale 24
Real 29
Strings 30
strukturierte 30
TypeOf 211
typisierte Konstanten 59

U
Übergabe
Adressen 223
von Funktionsergebnissen 225
von Parametern 223
Werte 224
Überlauf des Stacks
Compiler-Schalter 248
Umwandlung von Variablen 55
unäre Operatoren 67
UNDEF (bedingter Compiler-Befehl) 255
Unendlichkeiten 198
ungerade Zahlen 334
Units
Abhängigkeiten 124
gemeinsame Datendeklarationen 126
Implementationsteil 124
UnpackTime (Prozedur) 368
Unterbereiche 27
untypisierte Zeiger 42
UpCase (Funktion) 368

V
Val (Prozedur) 368
var-Parameter 223, 272
Strings, Compiler-Schalter 248
Var-string checking (Compiler-Schalter) 248
Variablen 49
absolute 51
Adress-Ermittlung 296
Bezug 52
Deklaration 49
DosError 157

dynamische 55
erhöhen 325
Erniedrigen 305
ErrorAddr 230
ExitCode 230
ExitProc 229
Funktionsergebnis 279
globale 291
globale 50
in der Unit System 150
in der Unit WinDos 153
Indizierungen 53
lokale 272, 291
lokale 50
Parameter 110
Parameter 223
Prozedur- 112
RandSeed 338
Records und Feldbezeichner 54
Speicherbedarf 354
Typ PChar 42
Typ Pointer 42
Typ-Umwandlung 55
Verbundanweisung 89
Vererbung 35
Vergleichsoperatoren 76
Verify (DOS-Flag) 353
Verzeichnisse
Erzeugen 331
Löschen 345
Prozeduren 158
Standard ermitteln 321
Wechseln 301
virtual (Schlüsselwort) 38
virtuelle Methode 38
Aufruf 40, 211
VMT 206
Aufbau 207
interne Verarbeitung 205
Zeigerinitialisierung 208
Parameter 213
Vorausberechnung von Konstanten 237

W
Wertebereiche
Prüfung 239
Wertparameter 110
Wertübergabe 224
WhereX (Funktion) 370
WhereY (Funktion) 370
while-Anweisung 93
Winkeleinheit Rad 304
with
Anweisung 96
with (Schlüsselwort) 55
Word (Datentyp)
als Parameter 224
Speicherung 196
WordBool (Datentyp) 26
Speicherung 197
wortweise Ausrichtung 240
Compiler-Schalter 244
Write (Prozedur) 370
für das Format Extended 186
Writeln
Prozedur 373
für das Format Extended 186

Z
Zahlensystem 269
Zeichenausgabe
über WinCrt 172
Zeichenketten 13
Zeichensatz-Tabelle 401ff.
Zeiger
@Result 279
als Parameter 224
auf Objekt 40
direkt setzen 337
normalisierte Form 77
Speicherung 200
typisierte Konstanten 64
untypisierte 42
Vergleiche 77
Zuweisungskompatibilität 40
Zufallswerte 338
Initialisierung 338
Zugriff
auf I/O-Adressen 236
Zuweisung 86
Kompatibilität 86
von Prozedur-Variablen 82

PROGRAMMIER-
HANDBUCH

TURBO PASCAL®

für Windows

BORLAND

BORLAND GMBH, LINDWURMSTRASSE 88, POSTFACH 15 03 40, D-8000 MÜNCHEN 2, TELEFON 0 89/7 20 10-0, TELEFAX 0 89/77 93 39, TELEX 5 212 637 mcm d,
CORPORATE HEADQUARTERS: 1800 GREEN HILLS ROAD, P.O. BOX 660001, SCOTTS VALLEY, CA 95067-0001. (408) 438-530. OFFICES IN:
AUSTRALIA, DENMARK, FRANCE, GERMANY, ITALY, JAPAN, NEW ZEALAND, SINGAPORE, SWEDEN AND THE UNITED KINDOM * ARTIKEL-NR. 112302